

Minimizing the Use of the Honest Majority in YOSO MPC with Guaranteed Output Delivery

Rishabh Bhadauria¹, James Chiang², Divya Ravi³, Jure Sternad², and Sophia Yakoubov

¹ Georgetown University

² Aarhus University

³ University of Amsterdam

Abstract. Cleve (STOC 86) shows that an honest majority is necessary for MPC with guaranteed output delivery. In this paper, we show that while an honest majority is indeed necessary, its involvement can be minimal. We demonstrate an MPC protocol with guaranteed output delivery, the majority of which is executed by a sequence of committees with dishonest majority; we leverage *one* committee with an honest majority, each member of which does work independent of the circuit size. Our protocol has the desirable property that every participant speaks only once (YOSO, Crypto 2021).

As a building block of independent interest, we introduce *public computation*, which is essentially privacy-free MPC with guaranteed output delivery (akin to smart contracts realized on blockchains). We instantiate public computation on a public bulletin board in three different ways (with different assumption / round / space utilization trade-offs).

1 Introduction

In today’s digital world, it is desirable for lightweight clients to outsource the evaluation of computationally demanding joint functions to service providers. For efficiency, it is essential that each client’s work remains independent of the function complexity (and, in particular, independent of the number of other clients also providing input). Often, a client might additionally require *privacy*, meaning that the service provider — and other clients — should learn nothing about the client’s input apart from what is revealed by the output.

For distributed service providers consisting of many heterogeneous machines, secure multi-party computation (MPC) [23,34] is a natural tool. As long as fewer than a corruption threshold t of the service provider machines are compromised, MPC protocols provide both client-side efficiency and privacy.

MPC protocols can be classified by the guarantees they offer, and by the assumptions they make on the adversary’s power to corrupt machines. The strongest guarantee an MPC protocol can make is *guaranteed output delivery*, where no action corrupt machines take can prevent honest participants from obtaining

the correct function output. Some MPC protocols offer their guarantees even in the event that a majority of the participants are under adversarial control. Unfortunately, Cleve [13] shows that MPC with guaranteed output delivery is unachievable in this dishonest majority setting.

“You Only Speak Once” (YOSO) MPC, first introduced by Gentry *et al* [20], is a class of MPC protocols where every participating machine speaks at most once. In a given round, a set of machines referred to as a “committee” speaks, after which that committee does nothing further, and the computation is picked up by a different committee in the following round. SCALES [1] is a related class of protocols where service provider machines are YOSO (meaning that they speak at most once), but clients can speak multiple times.

YOSO protocol design can serve two purposes. First, machines might come and go, and YOSO-style protocols allow machines to contribute to the computation without needing to commit their resources for long. Second, YOSO protocols enable us to keep communication complexity low in the presence of an adversary who has the power to compromise machines adaptively. By keeping committee members unpredictable, we can prevent an adversary from targeting them before they speak. If the adversary targets machines after they speak, by the time a machine is compromised that machine will no longer be relevant. In this way, a YOSO protocol can be designed to have total communication complexity sublinear in the number of available machines, despite the adversary’s ability to corrupt a linear number of those machines.

YOSO-style protocols, where a different committee speaks in every round, invite us to consider how much of the work truly needs to be done by honest majority committees in order to achieve guaranteed output delivery. Some YOSO protocols (e.g. the CDN-style construction of Gentry *et al* [20]) already leverage dishonest majority committees interspersed with the honest majority ones; however, all known protocols require a number of honest-majority committees that depends on the circuit depth [20,14], or require the honest-majority committees to do work proportional to the circuit size [28].

Our Contribution In this paper we show that a *single* honest-majority committee — performing computation *independent* of the function being evaluated (and in particular, independent of the number of inputs) — suffices to obtain guaranteed output delivery. In the SCALES setting, this honest-majority committee can be taken to be the set of clients; however, keeping the set of clients and honest-majority committee separate allows for a *dishonest* majority amongst the clients.

As a building block for our YOSO MPC protocol, we introduce public computation, which is essentially privacy-free MPC with guaranteed output delivery (akin to smart contracts). We believe public computation to be of independent interest. We instantiate public computation using a public bulletin board, in three YOSO-style ways: The first two use only three rounds each, and the third requires a logarithmic (in the size of the computation) number of rounds. Our first approach uses no cryptographic assumptions, but requires that participants post entire computation transcripts. The second uses strong cryptographic hardness assumptions (SNARKs), but reduces the amount of data posted from linear

in the size of the computation to polynomial in the size of the input. The third approach reaches an assumption / bulletin board space compromise: it relies only on collision-resistant hash functions, and uses a logarithmic (in the computation size) amount of space.

1.1 Related Work

		Clients Speak Once	Clients Speak More than Once
Guaranteed Output Delivery	Honest Majority of Servers	YOSO [20,28,8], Fluid [14]	
	Dishonest Majority of Servers (honest majority of clients)		this work
	Dishonest Majority of Servers (since Light-Weight Honest-Majority Committee)	this work	
Security With Abort	Honest Majority of Servers	Fluid [11,6]	
	Dishonest Majority of Servers	Fluid [32]	SCALES [1,2]

Table 1: Summary of YOSO, Fluid and SCALES Schemes. (Only protocols where servers speak once are included.)

YOSO MPC, motivated by security against adaptive adversaries, was introduced at the same time as Fluid MPC [11], a similar model motivated by dynamic availability of computing power. Since then, several YOSO and Fluid protocols have been described, with YOSO protocols primarily achieving guaranteed output delivery, and Fluid protocols focusing on *security with abort*, a weaker guarantee where compromised machines might cause a denial of service. In Table 1, we summarize existing YOSO, Fluid and SCALES constructions.

In the traditional, non-YOSO dishonest majority setting, a line of work by Choudhuri *et al* investigates MPC *fairness* with the aid of public bulletin boards [12]. Here, the impossibility of fairness without honest majority is overcome with the aid of a public bulletin board and stronger cryptographic primitives such as witness encryption [18].

1.2 Technical Overview

Our YOSO MPC protocol follows the template of threshold fully-homomorphic encryption (TFHE) based protocols (e.g. [3,24,7]).

Key Generation Phase: TFHE keys are generated.

Input Encryption Phase: Clients encrypt their inputs to the resulting TFHE public key.

Computation Phase: The circuit is evaluated homomorphically on the encrypted inputs.

Output Phase: The output is decrypted by a threshold of secret key holders.

We know that we need an honest majority *somewhere* in order to achieve guaranteed output delivery, thanks to the famous result of Cleve [13]. The crucial observation is that the honest majority is *only necessary in the last step* — the threshold decryption. In that step, an honest majority really is unavoidable: since we want guaranteed output delivery, we need decryption to work even if t key-holders don't participate, and if $t > \frac{n}{2}$, then the remaining $n - t < \frac{n}{2} < t$ key-holders are able to decrypt, which contradicts privacy.

This leaves us with two sets of designated parties: clients (a majority of whom may be corrupt) who hold input, and our single honest-majority committee in charge of decryption. We ensure that clients and honest-majority committee members can remain light-weight: the computation that each of them performs is independent of the circuit and input size.

We design a protocol in which the rest of the steps are carried out by a sequence of committees each of which has a dishonest majority. In order to ensure the correctness of each phase of the computation, we introduce a new primitive which we call *public computation*. Public computation is essentially MPC with no privacy: its only guarantees are correctness and liveness (output delivery). It is similar to verifiable computation [19], where a light-weight client can verify a circuit evaluated by a powerful server in time sub-linear in the circuit size. While verifiable computation typically relies on a secret client-side verification key, public computation is verifiable by any third party.

Public computation helps us ensure the correctness of the first three steps. The (dishonest majority) key generation committee produces zero knowledge proofs of correct key generation, which are verified via public computation. The clients then need to process only a small amount of data to extract the TFHE public key. They, in turn, produce zero knowledge proofs of correct encryption, which are verified before homomorphic evaluation on the ciphertexts, all done once again via public computation.

Public Computation Protocols Our public computation protocols all leverage a public bulletin board. We use this bulletin board to enable participants to accuse other participants of not following a protocol by pointing to locations on the bulletin board where there is an error. By specifying where on the bulletin board each participant should post their message, we allow verifiable accusations of message omission in addition to accusations of malformed messages.

Remark 1. A public bulletin board is a form of consensus, the implementation of which over traditional channels (e.g. blockchain-style) requires an honest majority. We view the public bulletin board as a basic resource (e.g., perhaps a

blockchain is available for our use) rather than a sub-protocol. Lifting reliance on the public bulletin board is an interesting and important open problem.

We describe three public computation protocols: the *naive* protocol, the *SNARK-based* protocol and the *bisection* protocol.

Naive Public Computation Protocol The naive protocol does not use any additional cryptographic hardness assumptions, and relies on two dishonest-majority committees. Each member of the first committee evaluates the entire circuit and posts a transcript of its evaluation to the bulletin board. Each member of the second committee checks each posted transcript, and posts a pointer to the first incorrect gate of every faulty evaluation. To extract the output, a party only needs to read and check the accusations before extracting the output from a transcript with no valid accusations against it. Since each accusation points to a constant-size error, and the size of the output itself does not depend on the circuit or input sizes, the complexity of the output-reader only depends (quadratically) on the committee size m .

SNARK-based Public Computation Protocol The naive protocol has the downside of posting entire circuit evaluation transcripts to the bulletin board, which uses a lot of space. In our SNARK-based protocol, members of the first committee instead compute and post the output together with a compact zero-knowledge proof (SNARK) that the output is correct. This is similar to zero-knowledge roll-ups [16], which leverage the succinctness of SNARKs to prove correct execution of committed blockchain transactions without explicit verification of the transactions themselves. However, the verification complexity of such a proof in [16] still depends on the input size. To lift the dependency of output extraction complexity on input size, we have members of the first committee also include a transcript of the evaluation of the verification circuit. The second committee then checks these circuits and posts accusations, much like they do in the naive protocol.

Bisection Public Computation Protocol The bisection protocol falls somewhere in between the naive protocol and the SNARK-based protocol, both in terms of assumptions and in terms of bulletin board space utilization. It uses Merkle tree commitments [31] (which rely on collision-resistant hashing), and bulletin board space dependant logarithmically on the circuit size. In the bisection protocol, members of the first committee post commitments to their circuit evaluation transcripts. Subsequent dishonest-majority committees engage in an interactive binary search for the first disagreeing locations in these transcripts; once found, these locations can be used to identify which of two disagreeing transcripts is erroneous. This interactive approach to resolve disputes is inspired from [15].

2 Preliminaries

2.1 YOSO Secure Multiparty Computation (MPC) Definitions

In this section we recap (verbatim from [28]) what it means for an MPC protocol to be YOSO secure. The YOSO model [20] makes a crucial separation between

physical machines and the roles which they play in the protocol. In this paper, we describe our YOSO MPC protocols in terms of roles. We ignore how roles are assigned to machines; we assume the availability of a role assignment functionality which allows point-to-point communication and broadcast messages between roles. Mechanisms which realize such a role assignment functionality were described by Benhamouda *et al* [5], Gentry *et al* [21] and Canetti *et al* [10]. The YOSO model uses the UC framework [9], with roles instead of physical machines as the participants. Every participant is ‘YOSO-ified’, meaning that as soon as she speaks for the first time, she is killed. A protocol Π YOSO-realizes a functionality \mathcal{F} if the YOSO-ification of Π UC-realizes \mathcal{F} 2.1.1.

2.1.1 UC MPC Consider a protocol $\Pi = (R_1, \dots, R_u)$ described as a tuple of roles R_i , each of which is a probabilistic polynomial-time (PPT) machine. Some of those roles are *input* roles, who, when they speak, provide an input. Other roles are there to assist in computing a function C on the provided inputs.

In a real-world execution of protocol Π with environment \mathcal{E} and adversary \mathcal{A} , the PPT environment \mathcal{E} provides the input $x = (x_1, \dots, x_\ell)$ to the protocol’s input roles. The environment also communicates with the PPT adversary \mathcal{A} . We consider a *synchronous* model, where the protocol is executed in rounds; in each round, some roles speak (or post to a public bulletin board). During the execution of the protocol, the corrupt roles receive arbitrary instructions from \mathcal{A} , while the honest roles faithfully follow the instructions of the protocol using the input they were given. We consider the adversary \mathcal{A} to be rushing, i.e., during every round the adversary can see the messages the honest roles sent before producing messages from corrupt roles. At the end of the protocol execution, the environment \mathcal{E} produces a binary output. Let $REAL_{\Pi, \mathcal{A}, \mathcal{E}}(1^{1^\lambda})$ denote the random variable (over the random coins used by all roles) representing \mathcal{E} ’s output in the real world.

Now, consider an ideal-world execution with the same environment \mathcal{E} , but with an ideal-world adversary \mathcal{S} . In the ideal-world execution, instead of running the protocol Π , the roles turn to a trusted party to compute C on the input given to them by \mathcal{E} . This trusted party receives the inputs x_1, \dots, x_ℓ from the input roles, and broadcasts $C(x_1, \dots, x_\ell)$. We call this trusted party the *ideal functionality* \mathcal{F}_{MPC} (described later in Section 3.3) for the computation of C with *guaranteed output delivery*. Let $IDEAL_{\mathcal{F}_{MPC}, \mathcal{S}, \mathcal{E}}(1^{1^\lambda})$ denote the random variable (over the random coins used by \mathcal{S}) representing \mathcal{E} ’s output in the ideal world.

Definition 1 (UC Security [9]). Let $C : (\{0, 1\}^*)^\ell \rightarrow \{0, 1\}^*$ be an ℓ -input function. A protocol $\Pi = (R_1, \dots, R_u)$ UC-securely computes C if for every PPT real-world adversary \mathcal{A} there exists a PPT ideal-world adversary (or simulator) \mathcal{S} such that, for any PPT environment \mathcal{E} , it holds that $REAL_{\Pi, \mathcal{A}, \mathcal{E}}(1^{1^\lambda})$ and $IDEAL_{\mathcal{F}_{MPC}, \mathcal{S}, \mathcal{E}}(1^{1^\lambda})$ are indistinguishable for any large enough security parameter 1^λ .

2.1.2 The Adversary’s Corruption Power Canetti *et al* [10] formalize a compiler that takes a YOSO MPC protocol described in terms of abstract roles and translates it to a YOSO MPC protocol described in terms of real machines. They show that the resulting protocol will be secure against an adversary with the ability to *adaptively* corrupt machines even if the original (abstract) protocol is only secure against an adversary who *statically* corrupts roles (i.e. decides which roles to corrupt before the protocol begins). We therefore focus on adversaries making static corruptions.

Our protocols proceed in rounds, where in every round, members of a committee speak. We assume that the adversary is able to corrupt a majority — but not everyone — on each committee. Our MPC protocol additionally relies on a single committee where only a minority of members may be corrupt.

3 Ideal Functionalities

In this section, we describe our ideal functionalities. The protocols in this paper are designed to offer *guaranteed output delivery*, meaning that no matter how corrupt parties misbehave, they cannot prevent protocol completion. Our ideal functionalities allow the simulator to trigger output delivery in order to model the passage of an appropriate number of rounds. This appears to allow the adversary to stall output delivery indefinitely, which contradicts our claim that our protocols deliver output no matter what. However, we recall that functionalities are meant to model safety guarantees but *not* liveness guarantees [9]; we instead argue guaranteed output delivery separately as a property of our protocols.

3.1 Public Bulletin Board (\mathcal{F}_{bb})

We consider the bulletin board in Fig. 1 as the only communication channel for honest parties. Following the approach of previous works [2,33,12], we assume that messages can be written and read at indexed locations, thus avoiding the need to read the entire bulletin board transcript to retrieve information. In particular,

1. Our bulletin board supports *random access* reads, allowing participants to access specific locations without reading the entirety of the bulletin board contents. This is important in order for select parties’ computation and communication complexity to remain independent of the amount of information posted.
2. Our bulletin board enforces a *gating function* for writes, ensuring that designated space on the board is only used by parties who are instructed by the protocol to post there.

While such a \mathcal{F}_{bb} functionality can be realized with an honest majority committee, we emphasize that we are not “offloading” MPC related work to this committee, or pushing complexities “down the stack” to the protocol realizing

\mathcal{F}_{bb} . The only computation required by \mathcal{F}_{bb} is the evaluation of the gating function which is polylogarithmic in the required protocol message locations when naively implemented as a table lookup: The number of required message locations is either linear ($\Pi_{\text{PubComp-Naive}}$) or logarithmic ($\Pi_{\text{PubComp-Bisection}}$) in the MPC circuit and input clients, implying a logarithmic sized index input. The gating function lookup itself would be logarithmic in the message locations.

If our bulletin board functionality is instantiated with a blockchain, such a polylogarithmic gating function can easily be added to the validator code or deployed as a smart contract. Given the complexity of the gating function outlined above, it is clear such validators are not contributing to the MPC computation itself by evaluating the gating function alone.

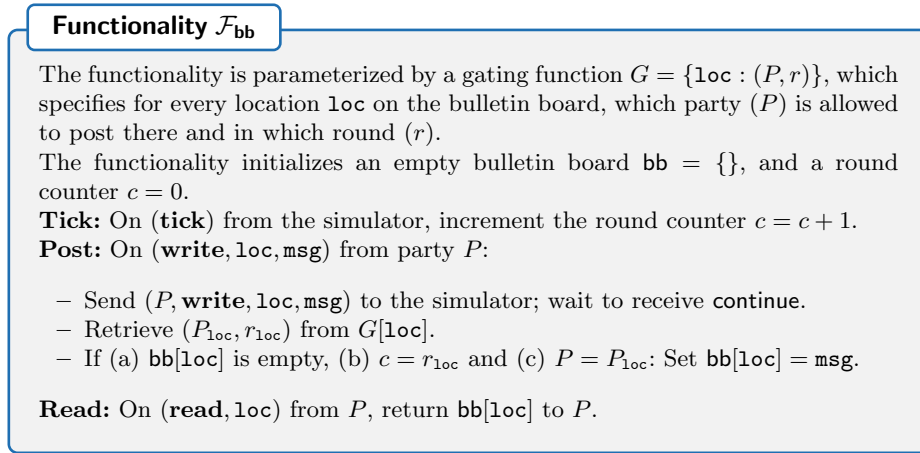
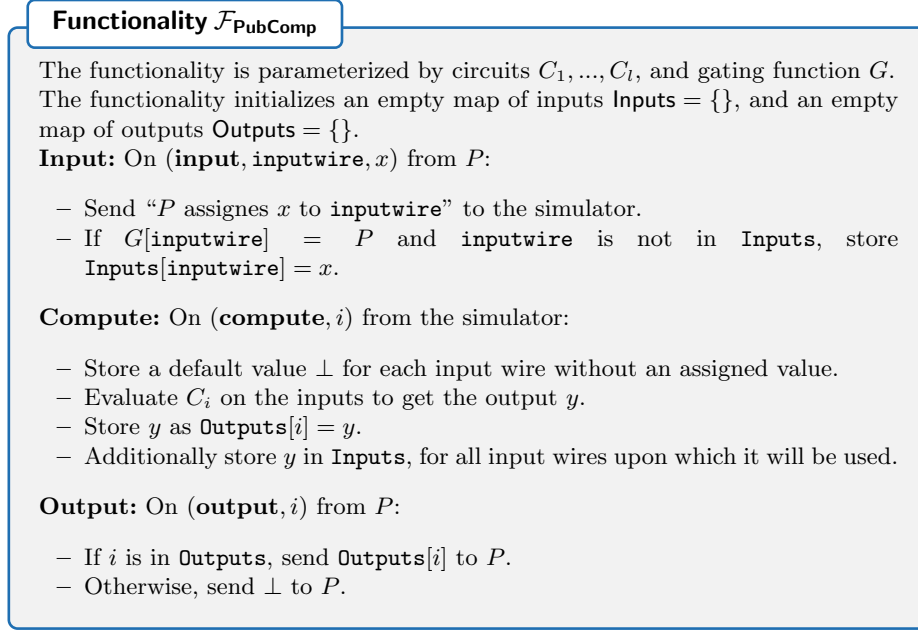


Fig. 1: Functionality \mathcal{F}_{bb}

3.2 Public Computation ($\mathcal{F}_{\text{PubComp}}$)

We define an ideal functionality $\mathcal{F}_{\text{PubComp}}$ in Fig. 2, which is parameterized by circuits C_1, \dots, C_l and gating function G . Let \mathcal{I} be the set of parties who hold inputs, **InputWires** be the set of input wires across all circuits, and **OutputWires** be the set of output wires. The gating function $G : \text{InputWires} \rightarrow \text{OutputWires} \cup \mathcal{I}$ specifies where inputs come from (outputs of other circuits, or input-holding parties). An input wire belonging to circuit i should take input from a party or from an output wire belonging to an earlier circuit ($< i$).

We abuse the notion of wires slightly; our wires do not carry bits, but instead carry entire values (e.g. ciphertexts, proofs, etc). Sometimes, for simplicity, multiple values are grouped on a single wire.

Fig. 2: Functionality $\mathcal{F}_{\text{PubComp}}$

Smart contracts and $\mathcal{F}_{\text{PubComp}}$: $\mathcal{F}_{\text{PubComp}}$ is similar to smart contracts realized by blockchain protocols. Both expose an interface which permits computation over public inputs. However, the simplicity of $\mathcal{F}_{\text{PubComp}}$ avoids many complications of UC functionalities [4,29] intended to accurately model the security of blockchain protocols. Additionally, blockchain protocols require an honest majority of servers to evaluate the public computation, whereas in this work, the computation is performed by members of dishonest majority committees only.

As detailed in Section 4, this is achieved by observing that the single honest server can always expose cheating behaviour by pointing towards a constant number of locations in \mathcal{F}_{bb} . In a nutshell, a computation server posting either the full computation transcript, a commitment thereof or a zero knowledge proof of correct computation, exposes itself to accusations by subsequent verification servers; such accusations can be verified by the output reader in time independent of the computation.

3.3 Secure Computation (\mathcal{F}_{MPC})

Finally, we define an MPC functionality \mathcal{F}_{MPC} in Fig. 3. It is parameterized by a circuit C and a gating function G , which specifies which input should be provided by which party.

Unlike public computation, which does not hide any of the inputs from the adversary, MPC is designed to keep honest parties' inputs private. So, \mathcal{F}_{MPC} does not forward the inputs themselves to the simulator.

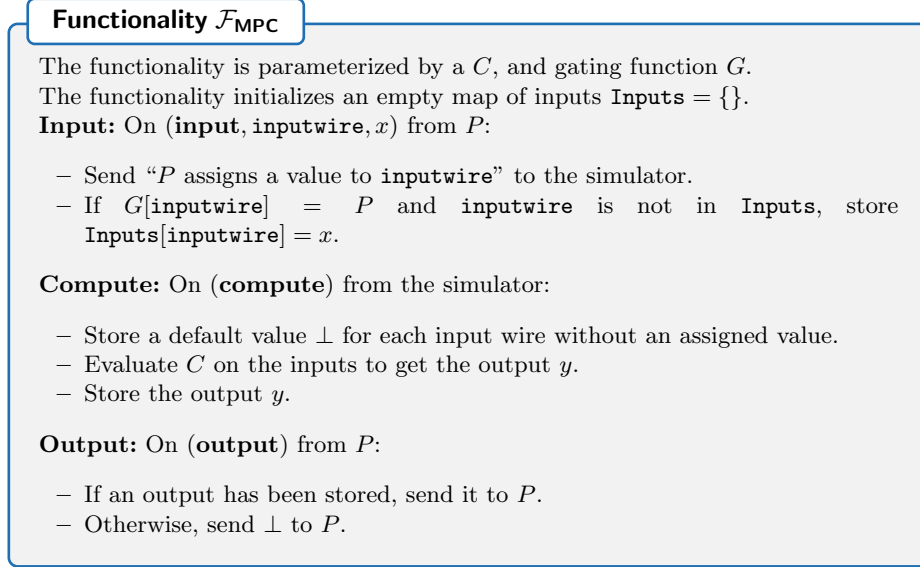


Fig. 3: Secure Computation (\mathcal{F}_{MPC})

4 Verifiable Computation Protocols

In this section we describe our instantiations of $\mathcal{F}_{\text{PubComp}}$. Our three protocols are $\Pi_{\text{PubComp-Naive}}$, $\Pi_{\text{PubComp-SNARG}}$, and $\Pi_{\text{PubComp-Bisection}}$.

$\Pi_{\text{PubComp-Naive}}$ uses no cryptographic assumptions and only requires three rounds, but takes up a lot of bulletin board space.

$\Pi_{\text{PubComp-SNARG}}$ uses SNARGs to reduce the amount of bulletin board space used, but the space complexity remains linear in the number of inputs to the computation being verified.

$\Pi_{\text{PubComp-Bisection}}$ is a compromise which requires more rounds and lighter assumptions (hash functions) while still using much less bulletin board space than the naive protocol. In contrast to $\Pi_{\text{PubComp-Naive/SNARG}}$, the \mathcal{F}_{bb} space complexity of $\Pi_{\text{PubComp-Bisection}}$ is *sub-linear* in both the number of computation inputs (and the overall computation), a desirable property in the setting of outsourced computation with many input providing clients.

Table 2 summarizes the trade-offs across the three constructions.

		$\Pi_{\text{PubComp-Naive}}$	$\Pi_{\text{PubComp-SNARG}}$ instantiated with [25]	$\Pi_{\text{PubComp-Bis-Pair}}$	$\Pi_{\text{PubComp-Bis-Runoff}}$
CS_i	Computation Complexity	$O(C)$	$O_\lambda(C \cdot \log(C))$	$O_\lambda(C)$	$O_\lambda(C)$
	\mathcal{F}_{bb} Write Complexity	$O(C)$	$O_\lambda(\ell)$	$O_\lambda(1)$	$O_\lambda(1)$
VS_i	Computation Complexity	$O(m C)$	$O_\lambda(m\ell)$	$O_\lambda(C + m^2 \log(C))$	$O_\lambda(C + m \log(C))$
	\mathcal{F}_{bb} Write Complexity	$O(m)$	$O(m)$	$O_\lambda(m^2 \cdot \log(C))$	$O_\lambda(m \log(C))$
Total \mathcal{F}_{bb} Write Complexity		$O(m C + m^2)$	$O_\lambda(m\ell) + O(m^2)$	$O_\lambda(m^3 \cdot \log(C)^2)$	$O_\lambda(m^2 \cdot \log(m) \cdot \log(C)^2)$
Output \mathcal{F}_{bb} Read Complexity		$O(m^2)$	$O(m^2)$	$O_\lambda(m^3 \cdot \log(C)^2)$	$O_\lambda(m^2 \cdot \log(m) \cdot \log(C)^2)$
Round complexity		3	3	$O(\log(C))$	$O(\log(C) \cdot \log(m))$
Assumptions		None	Trusted Setup (CRS) + SNARK	collision resistant hash (CRH)	collision resistant hash (CRH)

Table 2: Comparison of Public Computation Protocols. Let λ be the security parameter; m be the size of a dishonest-majority committee (e.g. there are m computation servers and m verification servers); $|C|$ be the size of the circuit, ℓ be the input size; and O_λ hides $\text{poly}(\lambda)$ factor. We assume that the output is of constant size.

4.1 Naive Protocol $\Pi_{\text{PubComp-Naive}}$

Our Naive protocol (Fig. 4) proceeds in three rounds: In the first round, input-holders provide their input via the bulletin board. In the second round, a dishonest-majority committee of servers which we will call *compute* servers posts messages to the bulletin board; in the third round, another dishonest-majority committee of servers which we will call *verification* servers posts. The computation output will be determined from the verification servers' (short) posts, and by (a small number of) random accesses to the rest of the bulletin board.

Our protocols make use of the bulletin board functionality \mathcal{F}_{bb} . As discussed in Section 3.1, \mathcal{F}_{bb} is parameterized by a gating function G_{bb} . We define the following bulletin board locations, the posting rules for which are hard-coded into G_{bb} :

- $\text{loc}_{\text{input},i}$: the location where P_i 's input should be written, in round 1.
- $\text{loc}_{\text{comp},i}$: the location where compute server CS_i should post in round 2. The compute servers may post a lot of data; we let $\text{loc}_{\text{comp},i,\text{outp}}$ be the location within $\text{loc}_{\text{comp},i}$ where the actual circuit output should be written.
- $\text{loc}_{\text{ver},i}$: the location where verification server VS_i should post in round 3.

Some of these locations need to be broken down into more specific addresses, for the sake of efficiency. We will handle this in prose.

The public computation functionality $\mathcal{F}_{\text{PubComp}}$ handles reactive computation by evaluating a sequence of circuits, each of which can take previous outputs or fresh values as input. In the interest of simplicity we describe our public computation protocols for a single circuit C .

4.1.1 Security

Theorem 1 (Security of $\Pi_{\text{PubComp-Naive}}$). *$\Pi_{\text{PubComp-Naive}}$ securely realizes $\mathcal{F}_{\text{PubComp}}$ if at least one computation server and at least one verification server is honest.*

Proof (Proof of Theorem 1). This protocol does not use any cryptographic primitives. We can specify a simulator which faithfully executes the role of honest input parties, at least one honest compute server, and at least one honest verification server. (It would advance the bulletin board round counter in between rounds.)

This is guaranteed to provide the correct output because the honest compute server — in both the real and the simulated execution — will post the correct function output. The honest verification server — in both the real and the simulated execution — will post valid accusations against all compute servers who did not compute correctly. No accusations against the honest compute server will verify, so the correct output can always be extracted.

4.1.2 Efficiency Let m be the number of verification servers (which is equal to the number of compute servers), and $|C|$ be the size of the circuit.

$\Pi_{\text{PubComp-Naive}}$

Input: In order to provide an input, party P_i sends $(\text{write}, \text{loc}_{\text{input}, i}, x_i)$ to \mathcal{F}_{bb} .

Compute: Computation proceeds in two rounds.

- **Computation:** Each computation server CS_i does the following:
 - Read all of the inputs from the bulletin board by sending $(\text{read}, \text{loc}_{\text{input}_1}), \dots, (\text{read}, \text{loc}_{\text{input}_\ell})$ to \mathcal{F}_{bb} . (If C takes previous outputs as input, those can similarly be read from the bulletin board.)
 - Evaluate C on the inputs. Let tscpt_i denote CS_i 's entire evaluation transcript, consisting of the values on all of the wires in the circuit.
 - Posts tscpt_i to the bulletin board by sending $(\text{write}, \text{loc}_{\text{comp}, i}, \text{tscpt}_i)$ to \mathcal{F}_{bb} .
- **Verification:** Each verification server VS_i does the following:
 - Initialize an empty set of accusations Accusations_i .
 - Read all of the inputs from the bulletin board.
 - Compute the circuit on the inputs to obtain the output y .
 - Read each compute server's output y_j from the bulletin board by sending $(\text{read}, \text{loc}_{\text{comp}, j, \text{outp}})$ to \mathcal{F}_{bb} .
 - * If $y_j = y$, move on to the next compute server.
 - * Otherwise, read CS_j 's entire transcript tscpt_j from the bulletin board by sending $(\text{read}, \text{loc}_{\text{comp}, j})$ to \mathcal{F}_{bb} . Check the transcript. Let $\text{loc}_{\text{err}, j}$ denote the location of the *first* erroneous gate in the transcript. Add $(j, \text{loc}_{\text{err}, j})$ to Accusations_i .
 - Post Accusations_i to the bulletin board by sending $(\text{write}, \text{loc}_{\text{ver}, i}, \text{Accusations}_i)$ to \mathcal{F}_{bb} .

Output: Party P does the following to retrieve the output of C :

- Read each verification server VS_i 's accusations by sending $(\text{read}, \text{loc}_{\text{ver}, i})$ to \mathcal{F}_{bb} .
- Check each of those accusations $(j, \text{loc}_{\text{err}, j})$ by sending $(\text{read}, \text{loc}_{\text{err}, j})$ to \mathcal{F}_{bb} and verifying that the gate found there was indeed evaluated incorrectly.
- Read the output from $\text{loc}_{\text{comp}, i, \text{outp}}$ for a compute server CS_i with no valid accusations against her.

Fig. 4: Protocol $\Pi_{\text{PubComp-Naive}}$

Bulletin Board Space The amount of bulletin board space used is dominated by (a) m transcripts of the circuit evaluation, which amounts to $O(m|C|)$ space, and (b) accusations posted by the verification servers which amounts to $O(m^2)$ space. The total space used is thus $O(m|C| + m^2)$.

Reading the Output To read the output, party P accesses $O(m)$ accusation messages on the bulletin board; each message is up to $O(m)$ in size, and points to up to $O(m)$ other (constant-size) locations. So, P must read $O(m^2)$ information from the bulletin board.

4.2 SNARG Protocol $\Pi_{\text{PubComp-SNARG}}$

In $\Pi_{\text{PubComp-SNARG}}$ (Fig. 5), we decrease the amount of data posted to the bulletin board by using SNARGs (Appendix A.2) to compress the transcript. We leverage two properties of SNARGs: proof succinctness, and verification efficiency (verification should be sublinear in the circuit size)⁴. Essentially, $\Pi_{\text{PubComp-SNARG}}$ runs $\Pi_{\text{PubComp-Naive}}$ on the SNARG verification circuit.

The computation servers use a SNARG to prove that the circuit C was correctly evaluated on the inputs (x_1, \dots, x_ℓ) . Our statement will consist of the inputs and output (x_1, \dots, x_ℓ, y) , and the witness of correct computation can be e.g. the computation transcript tsctpt . A statement, witness pair is considered to be in the relation if the witness is a valid transcript demonstrating that $y = C(x_1, \dots, x_\ell)$.

Let VC denote the verification circuit used to evaluate the SNARG verification algorithm Verify . Even though the CRS is linear in the circuit size ($O(|C|)$), the verifier needs to read only $O(\ell)$ pre-determined entries of the CRS. Therefore, in the verification circuit VC , we hardcode a small part of the CRS of size $O(\ell)$. Note that VC still has a linear dependency on the number of circuit inputs, implying that the space required for communication on the bulletin board will scale with the number of inputs; this dependency is overcome in $\Pi_{\text{PubComp-Bisection}}$ (Section 4.3) at the cost of additional protocol rounds.

4.2.1 Security

Theorem 2 (Security of $\Pi_{\text{PubComp-SNARG}}$). *$\Pi_{\text{PubComp-SNARG}}$ securely realizes $\mathcal{F}_{\text{PubComp}}$ if the SNARG used satisfies Soundness (defined in Definition 7), and if at least one computation server and one verification server is honest.*

Proof (Proof of Theorem 2). We can specify a simulator which faithfully executes the role of honest input parties, at least one honest compute server, and at least one honest verification server. (It would advance the bulletin board round counter in between rounds.) This is same as the simulator in the Theorem 1.

⁴ Pre-processing SNARGs give us both those properties. In our work, we focus on the SNARK presented by Groth16 [26] (every SNARK satisfies properties of SNARG), which provides a proof length $O_\lambda(1)$ and verification complexity of $O_\lambda(\ell)$ where ℓ is the size of the input and $O_\lambda(\cdot)$ hide $\text{poly}(\lambda)$ factors.

$\Pi_{\text{PubComp-SNARG}}$

Input: In order to provide an input, party P_i sends $(\text{write}, \text{loc}_{\text{input}, i}, x_i)$ to \mathcal{F}_{bb} .

Compute: Computation proceeds in two rounds.

- **Computation:** Each computation server CS_i does the following:
 - Read all of the inputs from the bulletin board by sending $(\text{read}, \text{loc}_{\text{input}_1}), \dots, (\text{read}, \text{loc}_{\text{input}_\ell})$ to \mathcal{F}_{bb} . (If C takes previous outputs as input, those can similarly be read from the bulletin board.)
 - Evaluate C on the inputs and obtains an output y_i . Let w_i be CS_i 's evaluation transcript.
 - Compute $\pi_i \leftarrow \text{Prove}(\text{crs}, \phi_i = (x_1, \dots, x_\ell, y_i), w_i)$.
 - Evaluate VC^a on (ϕ_i, π_i) . Let tscpt_i denote CS_i 's entire evaluation transcript, consisting of the values on all of the wires in the circuit VC .
 - Post (π_i, tscpt_i) to the bulletin board by sending $(\text{write}, \text{loc}_{\text{comp}, i}, (\pi_i, \text{tscpt}_i))$ to \mathcal{F}_{bb} .
- **Verification:** Each verification server VS_i does exactly what it would have done in $\Pi_{\text{PubComp-Naive}}$.

Output: Party P does exactly what it would have done in $\Pi_{\text{PubComp-Naive}}$.

^a VC is a circuit which emulates the algorithm $\text{Verify}(\text{crs}, \phi_i, \pi_i)$. Only a small part of crs is encoded in the circuit as the verifier does not need the whole CRS.

Fig. 5: Protocol $\Pi_{\text{PubComp-SNARG}}$

In the simulated execution, correct output is guaranteed because the honest compute servers will post the correct function output and the honest verification server will post valid accusations against all the compute servers who did not compute correctly. This is similar to the proof of Theorem 1.

In the real execution, the compute servers first compute the function, provide a SNARG proof and the verification transcript of the SNARG proof. The only case where the real execution deviates from the simulated execution is when a dishonest compute server generates a SNARG proof for fake output \hat{y} and provides an incorrect SNARG which gets accepted by the Verify algorithm of SNARG. No verification server will accuse this dishonest server and two outputs will exist, breaking correctness. The SNARG soundness property (Definition 7) ensures that a valid proof is generated for a statement not in the language (or in our case, for a wrong output) only with negligible probability.

4.2.2 Efficiency Let m be the number of verification servers (which is equal to the number of compute servers), $|C|$ be the size of the circuit and ℓ be the total size of all the inputs put together.

Bulletin Board Space The amount of bulletin board space used is dominated by m SNARG proofs and m transcripts of the verification circuit. If the proof size is $O(|\pi|)$ and the verification computation is $O(v)$, then the amount of bulletin board space posted by the compute committees is $O(m(|\pi| + v))$. In our work, we plug in the SNARK due to Groth [25], which has a proof size $O_\lambda(1)$ and verification complexity $O_\lambda(\ell)$, reducing our bulletin board complexity to $O_\lambda(m\ell)$. The verification servers additionally post $O(m)$ size messages, using $O(m^2)$ bulletin board space; the total amount of bulletin board space used is thus $O_\lambda(m\ell) + O(m^2)$.

Reading the Output As the output stage is identical to $\Pi_{\text{PubComp-Naive}}$, party P reads only $O(m^2)$ information from the bulletin board.

4.3 Bisection protocol $\Pi_{\text{PubComp-Bisection}}$

In $\Pi_{\text{PubComp-Bisection}}$, we further decrease the amount of data posted by leveraging interaction, as inspired by [15]. The protocol starts with several servers computing the circuit we wish to evaluate, and committing to their entire computation transcript. We begin with a simplified case where only two persistent servers participate, which we formalize as $\Pi_{\text{PubComp-Bis-2PC-Persist}}$ in Section 4.3.1.

Looking forward, while this protocol executed by two persistent computation servers is non-YOSO, it serves as a stepping-stone towards a YOSO-fied version in Section 4.3.1 for ephemeral committees and 2 commitments. Finally, we present two general YOSO protocols $\Pi_{\text{PubComp-Bis-Pair}}$ and $\Pi_{\text{PubComp-Bis-Runoff}}$ for m commitments and ephemeral servers in Section 4.3.3. We use $\Pi_{\text{PubComp-Bis-2PC}}$ as a subroutine.

4.3.1 Bisection Protocol with Two Transcript Commitments We start with two computation servers CS_1 and CS_2 . Each server evaluates the circuit. To avoid explicitly posting each wire assignment to \mathcal{F}_{bb} , we require both servers to post a Merkle commitment of the circuit evaluation transcript. The transcript will consist of (input or gate output) wire value assignments $w^1, \dots, w^{|C|}$, for a unique topological wire ordering⁵ of C 's gates, where the last wire value $w^{|C|} = y$ corresponds to the output.

Given that the topological ordering of wires is uniquely defined, two differing evaluations of the circuit will result in inconsistent Merkle roots. We will exploit this to guarantee the elimination of at least one commitment by communicating only a logarithmic amount of data to \mathcal{F}_{bb} .

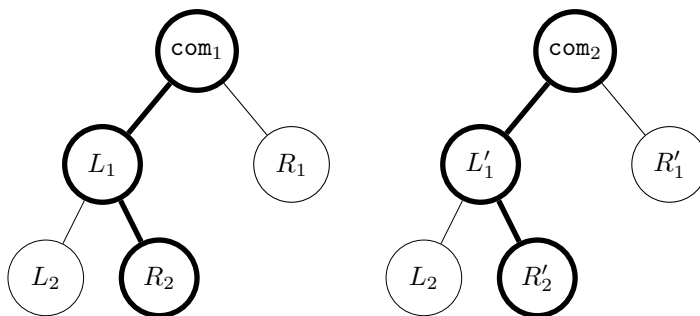


Fig. 6: In these two Merkle trees with distinct roots, the difference between com_1 and com_2 indicates an inconsistency in the committed wire assignments. To identify the first disagreement in the topologically ordered transcripts, the verification servers then open the roots in the second round, exposing $L_1 \neq L'_1$ and $R_1 \neq R'_1$, which indicates disagreements in both halves of committed transcripts. In the third round, the verification servers open the children of L_1 and L'_1 , exposing the first disagreeing leaves $R_2 \neq R'_2$ of the topologically ordered transcripts committed to com_1 and com_2 .

For two inconsistent commitments $com_1 \neq com_2$ to differing evaluations, there must exist a *first* Merkle tree leaf position corresponding to a wire in C at which com_1 and com_2 commit to different values. If this is an input wire, one of the transcripts must have used an incorrect input; if this is a gate output wire, one of the transcripts must contain an incorrect evaluation of that gate (since this is the first inconsistency, and thus the two transcripts must agree on the *inputs* to that gate). Given the opening of such a leaf position for both com_1 and com_2 , an observer can identify at least one of the two committed transcripts as incorrect.

⁵ An arithmetic circuit always has a topological ordering, which follows from its interpretation as a DAG, for which a topological ordering always exists. To ensure that the topological ordering for any circuit is uniquely defined, we simply assume a concrete, pre-defined deterministic sorting algorithm for this purpose (e.g. [27]).

Finding this leaf position via interactive binary search gives a public computation protocol for two parties, at least one of which is honest. We describe this protocol $\Pi_{\text{PubComp-Bis-2PC-Persist}}$ (Fig. 7), where two parties CS_1 and CS_2 (each given the entire tree corresponding to com_1 and com_2 respectively) find the first index of their conflict. In each round of $\Pi_{\text{PubComp-Bis-2PC-Persist}}$, CS_1 and CS_2 post the child nodes of the same single internal node position for each of the Merkle tree commitments.

$\Pi_{\text{PubComp-Bis-2PC-Persist}}$ makes use of the following bulletin board locations:

$\text{loc}_{\text{input},i}$ is where Client_i posts their input.

$\text{loc}_{\text{commit},i}$ is where CS_i posts their Merkle tree root.

$\text{loc}_{\text{ver},r,i}$ is where CS_i posts their r th pair of Merkle tree nodes.

$\text{loc}_{\text{accuse},i}$ is where CS_i posts their accusation against CS_j .

Towards adapting $\Pi_{\text{PubComp-Bis-2C-Persist}}$ for ephemeral, stateless servers which only speak once, we highlight that any cheating in $\Pi_{\text{PubComp-Bis-2C-Persist}}$ is already publicly observable by reading $O_\lambda(\log(|C|))$ bits on \mathcal{F}_{bb} . This facilitates the introduction of dedicated verification servers, which can show up, verify and extend the state of the protocol execution from bulletin board messages alone.

Lemma 1. *Adversarial behaviour in $\Pi_{\text{PubComp-Bis-2C-Persist}}$ is publicly detectable by reading at most $O_\lambda(\log(|C|))$ bits from \mathcal{F}_{bb} .*

Proof. (Sketch) The deterministic computation performed by parties on public inputs on \mathcal{F}_{bb} is fixed. Once a party has posted the merkle tree root of its evaluation transcript, it has committed to all the internal nodes it can open during the verification phase, or otherwise breaks collision resistance of the hash function. As a consequence, the path that is traversed during $\Pi_{\text{PubComp-Bis-2C-Persist}}$ is also fixed, as this is solely a function of the internal node values in the merkle trees of both parties.

Once the adversary has posted an incorrect (y_i, com_i) during the computation phase, any adversarial strategy is restricted to

- Case 1. Completing the rest of the protocol honestly
- Case 2. Deviating from the protocol by
 - a. Sending an invalid message.
 - b. Remaining silent.

Case 1: Completing the protocol honestly after posting an incorrect (y, com) will either expose the first wire assignment which differs to that of the honest party, hereby exposing the incorrect computation transcript, or if com is well-formed, expose the last committed output wire (wire assignments are in topological order), which is inconsistent with y posted by the adversary in the computation phase.

Case 2a: Extending the merkle tree opening along an incorrect path with valid openings of internal nodes implies breaking collision resistance. Otherwise, an invalid opening of a node in the merkle tree is publicly observable once it is posted to \mathcal{F}_{bb} .

$\Pi_{\text{PubComp-Bis-2C-Persist}}$

Input: In order to provide an input, party P_i sends $(\text{write}, \text{loc}_{\text{input}, i}, x_i)$ to \mathcal{F}_{bb} .

Compute:

- **Computation:** Each computation server CS_i for $i \in [1, 2]$ does the following:
 - Read all of the inputs from the bulletin board by sending $(\text{read}, \text{loc}_{\text{input}, 1}), \dots, (\text{read}, \text{loc}_{\text{input}, \ell})$ to \mathcal{F}_{bb} .
 - Evaluate C on the inputs and obtain an output y_i . Let tscpt_i be CS_i 's evaluation transcript with the gate wire assignments in topological order of the circuit.
 - Compute $\text{com}_i \leftarrow \text{Commit}(\text{tscpt}_i)$.
 - Post (y_i, com_i) to the bulletin board by sending $(\text{write}, \text{loc}_{\text{commit}, i}, (y_i, \text{com}_i))$ to \mathcal{F}_{bb} .
 - Let $\text{curNode}_i = \text{com}_i$.
 - Read the other party's commitment curNode_j from the bulletin board by sending $(\text{read}, \text{loc}_{\text{commit}, j})$ to \mathcal{F}_{bb} .
- **Verification:** For each round $r \in [2, \dots, \log(|C|)]$, each server CS_i for $i \in [1, 2]$ does the following:
 - Let $L_{r,i}, R_{r,i}$ be the children of curNode_i in CS_i 's Merkle tree.
 - Send $(\text{write}, \text{loc}_{\text{ver}, r, i}, (\text{curNode}_i, L_{r,i}, R_{r,i}))$ to \mathcal{F}_{bb} .
 - Read the other party's children $(L_{r,j}, R_{r,j})$ from the bulletin board by sending $(\text{read}, \text{loc}_{\text{ver}, r, j})$ to \mathcal{F}_{bb} .
 - If $\text{Commit}(L_{r,j}, R_{r,j}) \neq \text{curNode}_j$: accuse the other party by writing a pointer to the inconsistency at $\text{loc}_{\text{accuse}, i}$ on the bulletin board.
 - Otherwise:
 - * If $L_{r,i} \neq L_{r,j}$: set $\text{curNode}_i = L_{r,i}$ and $\text{curNode}_j = L_{r,j}$.
 - * Otherwise: set $\text{curNode}_i = R_{r,i}$ and $\text{curNode}_j = R_{r,j}$.

In round $r = \log(|C|) + 1$, we should have reached the leaves of our Merkle trees. Each server CS_i for $i \in [1, 2]$ does the following:

- Write the leaf node to $\text{loc}_{\text{ver}, r, i}$. If the leaf node represents a gate output wire, also decommit the two input wires for that gate.
- Accuse the other party by writing a pointer to their leaf node. If the leaf node represents a gate output wire, the other party's gate evaluation must be faulty, since the two evaluations agree on all prior wire values.
- If $\text{com}_1 = \text{com}_2$, but $y_1 \neq y_2$, accuse the other party by posting an opening of the last wire (y_i) and a pointer to their output.

Output: Party P checks both accusations, and reads the output from $\text{loc}_{\text{input}, i}$ for CS_i without a valid accusation against it.

Fig. 7: Protocol $\Pi_{\text{PubComp-Bis-2C-Persist}}$

Case 2b: Silence by a party is publicly observable by reading a single, dedicated location for the specific party and round on \mathcal{F}_{bb} .

Posting a correct (y, com) during the computation phase restricts any subsequent adversarial behaviour to sending an invalid message or remaining silent, both which can be detected with location lookups on \mathcal{F}_{bb} .

The entire protocol transcript occupies at most $O(\log(|C|) \cdot \text{poly}(\lambda))$ bits on \mathcal{F}_{bb} ; the protocol specifies $O(\log(|C|))$ internal node openings of $\text{poly}(\lambda)$ size.

4.3.2 Bisection Protocol with Two Commitments: the YOSO Version

Our next challenge is to modify this two-server protocol so that no server is required to speak more than once (Fig. 8). We can do this by making a simple observation: our public computation has no private inputs, and the state of an honest server can be deterministically recomputed by any other honest party.

We introduce $\log(|C|)$ committees, where the committee for round $r \in [2, \dots, \log(|C|) + 1]$ consists of m verification servers $\text{VS}_{r,1}, \dots, \text{VS}_{r,m}$. Each server $\text{VS}_{r,i}$ posts to $\text{loc}_{\text{ver},r,i}$. In each round r , server $\text{VS}_{r,i}$ computes the circuit C anew, determines which prior round $r - 1$ servers it supports, and publishes a pointer to their bulletin board posts along with its own $(L_{r,i}, R_{r,i})$, extending the path of nodes opened by the prior $r - 1$ servers that $\text{VS}_{r,i}$ deems honest.

We finish with an accusation committee. Accusations are a bit involved, since each round comprises m posts and multiple servers might support the same Merkle tree root. An accusation now points out that in an *entire round*, a given Merkle tree root has *no* valid support. Validating an accusation of no support can be achieved with m location lookups on \mathcal{F}_{bb} and concluding that the internal path of a merkle tree was not extended correctly by any party.

In order to extract output, P will only have to read $O(m^2 + \log(|C|))$ data from the bulletin board: P will need to go through m accusations, each of which will point at a m -sized round. In order to verify gates, it will need to verify entire $\log(|C|)$ -sized paths through the Merkle trees (to validate the gate inputs).

4.3.3 General YOSO Bisection Protocol

The binary search style nature of the bisection protocol does not obviously generalize to a higher number m of disagreeing Merkle roots. We could run the two-root protocol $\Pi_{\text{PubComp-Bis-2C}}$ in parallel for every pair of roots; this would yield bulletin board and extraction complexities that are $\binom{m}{2} = \frac{m(m-1)}{2}$ times those of the two-root protocol. This is formalized in Fig. 9.

Here, we emphasize that in each $\binom{m}{2}$ parallel execution of the two-root protocol, it is no longer given that one of the commitment roots is honest. Thus, the honest verifier $\text{VS}_{r,i}$ which recomputes the circuit C correctly will not extend internal nodes of any two-root subprotocol instance which does not include a correct root. Nonetheless, the honest server can actively provide accusations for any publicly detectable cheating (Lemma 1) for all $O(m^2)$ two-root executions. A root which obtains *no valid accusations* in all two-root executions is output by the output reader.

$\Pi_{\text{PubComp-Bis-2C}}$

Input: In order to provide an input, party P_i sends $(\text{write}, \text{loc}_{\text{input}, i}, x_i)$ to \mathcal{F}_{bb} .

Compute:

- **Computation:** Each computation server CS_i for $i \in [1, 2]$ performs computation as in $\Pi_{\text{PubComp-Bisection-2PC-static}}$, but terminates after posting the output y_i and evaluation transcript commitment com_i to the bulletin board.
- **Verification:** For each round $r \in [1, \dots, \log(|C|)]$, each verification server VS_i performs the following:
 - Perform the same steps as computation servers to obtain (y_i, com_i) .

For round $r = 1$, each VS_i then performs the following:

- Let $L_{1,i}, R_{1,i}$ be the children of com_i . VS_i sends $(\text{write}, \text{loc}_{\text{ver}, 1, i}, (\text{com}_i, L_{1,i}, R_{1,i}))$ to \mathcal{F}_{bb} .

For each $r \in [2, \dots, \log(|C|)]$, each VS_i then performs the following:

- Read the computation messages sent by each computation server CS_j by sending $(\text{read}, \text{loc}_{\text{comp}, j})$ to \mathcal{F}_{bb} .
- Read all verification messages sent by each verification server VS_j in prior rounds by sending $(\text{read}, r' < r, \text{loc}_{\text{ver}, j})$ to \mathcal{F}_{bb} .
- VS_i replays all prior verification rounds.

For $r' \in [1, \dots, r - 1]$

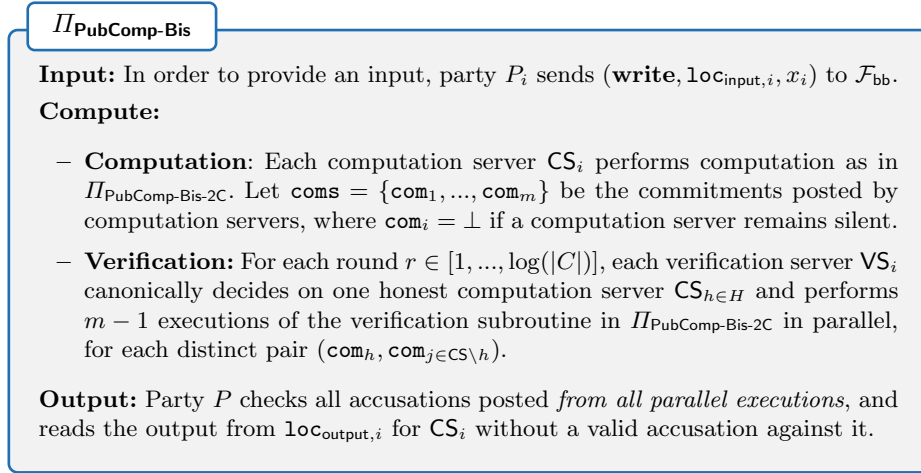
- * If $r' = 1$, set $\text{currNode}_i = \text{com}_i$ and $\text{currNode}_j = \text{com}_j$, such that $\{\text{com}_i, \text{com}_j\}$ is equal to the set of commitments posted by the two computation servers.
 - If only one computation commitment was posted, send an accusation of the silent computation server CS_j by sending $(\text{write}, \text{loc}_{\text{accuse}, r, i}, j)$ to \mathcal{F}_{bb} and terminate.
- * Let $L_{r', i}, R_{r', i}$ be children of currNode_i computed by VS_i .
- * Let $L_{r', j}, R_{r', j}$ be children of currNode_j posted by prior verification servers.
 - If no valid children were posted to \mathcal{F}_{bb} by prior verification servers, point out the lack of support for the CS_j which posted the merkle root of currNode_j by sending $(\text{write}, \text{loc}_{\text{accuse}, r, i}, \text{CS}_j)$ to \mathcal{F}_{bb} and terminate.
- * If $L_{r', i} \neq L_{r', j}$, set $\text{currNode}_i \neq L_{r', i}$.
- * Otherwise, set $\text{currNode}_i \neq R_{r', i}$.
- VS_i sends $(\text{write}, \text{loc}_{\text{ver}, r, i}, (L_{r-1, j}, (L_{r, i}, R_{r, i})))$ to \mathcal{F}_{bb} .

In round $r = \log(|C|) + 1$, each VS_i follows the same steps as in the last round of $\Pi_{\text{PubComp-Bis-2C-Persist}}$ and writes accusations pointing to invalid wires.

Output: Party P checks all accusations, and reads the output from $\text{loc}_{\text{input}, i}$ for CS_i without a valid accusation against it.

Fig. 8: Protocol $\Pi_{\text{PubComp-Bis-2C}}$

We note an optimization implemented in $\Pi_{\text{PubComp-Bis}}$ (Fig. 9) which does not require the full complexity of $\binom{m}{2}$ bisection protocol instances. Here, the honest servers will only run the bisection protocol for $m - 1$ pairs of servers. For simplicity, assume CS_i is honest. Then honest verification servers will run the bisection protocol for the commitment and output from computation server pairs $(\text{CS}_i, \text{CS}_1), (\text{CS}_i, \text{CS}_2), \dots, (\text{CS}_i, \text{CS}_m)$. Each cheating CS_j will receive valid accusations in the process of running the bisection protocol for these $m - 1$ pairs of computation commitments. If there are multiple honest computation servers that post correctly computed outputs and commitments, the verification servers can canonically select, for example, one with the highest index and match its commitment against all other commitments in the same fashion.

Fig. 9: Protocol $\Pi_{\text{PubComp-Bis}}$

4.3.4 Security

Theorem 3 (Security of $\Pi_{\text{PubComp-Bis}}$). $\Pi_{\text{PubComp-Bis}}$ securely realizes $\mathcal{F}_{\text{PubComp}}$ as long as at least one computation server and at least one verification server is honest.

Proof (Proof of Theorem 3). The simulator follows that of Thm. 1.

The simulator fails if the simulated public computation protocol cannot deliver a definitive, publicly verifying output. This occurs when the simulated bisection protocol cannot resolve two inconsistent (commitment, output) pairs, of which one is honestly computed. In this case, the path of internal node openings follows the protocol correctly, but results in two opened leaves that are identical, valid wire assignments - since the computation is deterministic, this implies a

violation of the collision resistance property of the hash function with which the Merkle tree is instantiated.

4.3.5 Efficiency

Bulletin Board Space First, we analyze the efficiency of $\Pi_{\text{PubComp-Bis-2PC}}$. Each of the two compute servers posts a single hash, which takes $O_\lambda(1)$ space. Each verification server posts at most $O_\lambda(\log(|C|))$ data (e.g. when having to open input wires to a gate). After $\log(|C|)$ rounds where m servers post in each round, this amounts to $O_\lambda(m \log(|C|)^2)$ data on the bulletin board.

In $\Pi_{\text{PubComp-Bis}}$, we generalize the protocol to m compute servers instead of two. Running $\Pi_{\text{PubComp-Bis-2PC}}$ pairwise for $m - 1$ pairs induces an increase in all complexities by factor m .

Reading the Output The output is extracted from the entire bulletin board transcript.

5 MPC Protocol

In this section we instantiate \mathcal{F}_{MPC} . Our MPC protocol uses threshold fully homomorphic encryption (TFHE, Appendix A.1), succinct non-interactive zero-knowledge (zkSNARK, Appendix A.2), and public computation $\mathcal{F}_{\text{PubComp}}$.

Our protocol proceeds in several phases:

Key Generation: A dishonest-majority committee of key generation servers K_1, \dots, K_m generates the TFHE keys. $\mathcal{F}_{\text{PubComp}}$ checks that the keys were generated correctly (by checking the zero knowledge proofs provided by the key generation servers).

Input: The clients $\text{Client}_1, \dots, \text{Client}_\ell$ encrypt their inputs to the (aggregated) TFHE public key. $\mathcal{F}_{\text{PubComp}}$ checks the input ciphertexts (by checking the zero knowledge proofs provided), and homomorphically evaluates the circuit C on the encrypted inputs.

Output Computation: An honest-majority committee of decryption servers D_1, \dots, D_n jointly decrypt the output. $\mathcal{F}_{\text{PubComp}}$ checks that the partial decryptions were produced correctly (by checking the zero knowledge proofs provided), and aggregates the partial decryptions to produce the final output.

5.1 Relations for Zero Knowledge Proofs

Zero knowledge proofs are used thrice in the protocol: the key generation servers prove that they did their job correctly, the clients prove that their input ciphertexts are well-formed, and the decryption servers prove that they produced their partial decryptions correctly. We formalize the relations for those three zero knowledge proofs as follows:

$$\mathcal{R}_{\text{KGen}} = \left\{ \begin{array}{l} \phi = (ppk_1, \dots, ppk_n) \\ \quad (tpk, ct_1, \dots, ct_n) \\ w = \rho \end{array} \middle| \begin{array}{l} (tpk, \{tsk_i\}_{i \in [n]}) \leftarrow \text{TFHE.KGen}(\rho) \\ ct_i \leftarrow \text{Enc}(ppk_i, tsk_i; \rho) \end{array} \right\}$$

The witness for $\mathcal{R}_{\text{KGen}}$ is the randomness ρ . We abuse notation slightly by using ρ both for the generation of the TFHE keys, and for the encryption of the secret keys to the decryption committee. We assume that ρ is sufficiently long to enable the use of separate, independent parts of it for each operation.

$$\mathcal{R}_{\text{Enc}} = \left\{ \begin{array}{l} \phi = (tpk, ct) \\ w = (x, \rho) \end{array} \middle| ct \leftarrow \text{TFHE.Enc}(tpk, x; \rho) \right\}$$

$$\mathcal{R}_{\text{PDec}} = \left\{ \begin{array}{l} \phi = (\{ct_j\}_{j \in [L]}, ct, d, tpk) \\ w = psk \end{array} \middle| \begin{array}{l} \{tsk_j\}_{j \in [L]} \leftarrow \{\text{Dec}(psk, ct_j)\}_{j \in [L]} \\ d \leftarrow \text{TFHE.PDec}(tpk, \{tsk_j\}_{j \in [L]}, ct) \end{array} \right\}$$

5.2 Circuits for Public Computation

We invoke our public computation functionality $\mathcal{F}_{\text{PubComp}}$ thrice: to check and aggregate the generated keys, to check the input ciphertexts and homomorphically evaluate the circuit C , and to check and aggregate the partial decryptions. Formally, these computations require some inputs which are part of the setup. These inputs include the common reference strings crs_{KGen} , crs_{Enc} , crs_{PDec} , the encryption keys ppk_1, \dots, ppk_n belonging to the honest-majority decryption committee, and the circuit C representing the function which our MPC aims to evaluate. For simplicity, we assume that these are hardcoded into the circuits evaluated by $\mathcal{F}_{\text{PubComp}}$.

We specify the three circuits C_{KGen} , C_{Eval} and C_{Dec} evaluated by $\mathcal{F}_{\text{PubComp}}$ as follows:

- $C_{\text{KGen}}(\{\mathbf{tpk}_i, \mathbf{ct}_{i \rightarrow 1}, \dots, \mathbf{ct}_{i \rightarrow n}, \pi_i\}_{i \in [m]}) :$
- Initialize a list L_{KGen} to keep track of verifying proofs.
 - For $i \in [m]$:
 - Let $\phi_i = (ppk_1, \dots, ppk_n, tpk_i, ct_{i \rightarrow 1}, \dots, ct_{i \rightarrow n})$.
 - Check π_i by running $\text{Verify}(\text{crs}_{\text{KGen}}, \phi_i, \pi_i)$. If it verifies, add i to L_{KGen} .
 - Aggregate the verifying TFHE public keys as $tpk \leftarrow \text{AggregateKeys}(\{tpk_i\}_{i \in L_{\text{KGen}}})$.
 - Return $(tpk, \{ct_{i \rightarrow j}\}_{i \in L_{\text{KGen}}, j \in [n]})$.
- $C_{\text{Eval}}(tpk, \{\mathbf{ct}_i, \pi_i\}_{i \in [\ell]}) :$
- For $i \in [\ell]$:
 - Check π_i by running $\text{Verify}(\text{crs}_{\text{Enc}}, \phi_i = (tpk, ct_i), \pi_i)$.
 - If it does not verify, replace ct_i with an encryption of a default value.
 - Output $ct = \text{Eval}(tpk, C, ct_1, \dots, ct_\ell)$.
- $C_{\text{Dec}}(tpk, \{ct_{j \rightarrow i}\}_{j \in L_{\text{KGen}}, i \in [n]}, ct, \{\mathbf{d}_i, \pi_i\}_{i \in [n]}) :$
- Initialize a list L_{Dec} to keep track of verifying proofs.

- For $i \in [n]$:
 - Check π_i by running $\text{Verify}(\text{crs}_{\text{PDec}}, \phi_i = (\{ct_{j \rightarrow i}\}_{j \in L_{\text{KGen}}}, ct, d), \pi_i)$.
If it verifies, add i to L_{Dec} .
- Output $y = \text{TFHE.Dec}(tpk, \{d_i\}_{i \in L_{\text{Dec}}})$.

The input wires for these circuits are specified as $(\text{KGen}, 1), \dots, (\text{KGen}, m), (\text{Enc}, 1), \dots, (\text{Enc}, \ell), (\text{Dec}, 1), \dots, (\text{Dec}, n)$. Values on these wires are denoted above in **blue**.

The other inputs to C_{Eval} and C_{Dec} are outputs of previous circuits. Both C_{Eval} and C_{Dec} use tpk , which is an output of C_{KGen} . Additionally, C_{Dec} uses $\{ct_{i \rightarrow j}\}_{i \in L_{\text{KGen}}, j \in [n]}$, which is output by C_{KGen} , and ct , which is output by C_{Eval} .

5.3 The Protocol

We next describe our MPC protocol (Fig. 10). We assume that the following is available prior to the execution of the protocol:

- The common reference strings $\text{crs}_{\text{KGen}}, \text{crs}_{\text{Enc}}$ and crs_{PDec} .
- The encryption keys ppk_1, \dots, ppk_n belonging to decryption servers.
- The circuit C to be computed.

Our protocol interacts with $\mathcal{F}_{\text{PubComp}}$ which is parameterized by the circuits described above. $\mathcal{F}_{\text{PubComp}}$ expects:

- Inputs to C_{KGen} from K_1, \dots, K_m ,
- Inputs to C_{Eval} from the clients $\text{Client}_1, \dots, \text{Client}_\ell$, and
- Inputs to C_{Dec} from D_1, \dots, D_n .

Some parties need to access *part* of circuit outputs. The complexity of these accesses scales with the accessed data, not with the size of the entire output.

In the above protocol, the output is assumed to be public. However, it is possible to extend it to private outputs using the following generic transformation: Each client includes a random value (a mask) as an additional input. The circuit computing the private outputs is now modified to compute a masked output for each client instead. Each client retrieves and unmask their own masked output.

5.4 Efficiency Analysis

Each client (1) reads a TFHE public key from $\mathcal{F}_{\text{PubComp}}$, (2) performs a TFHE encryption, (3) computes a zk-SNARK proof, and (4) inputs the ciphertext and proof back to $\mathcal{F}_{\text{PubComp}}$. Since the complexity of reading part of an output from $\mathcal{F}_{\text{PubComp}}$ should scale with the size of the part, not with the size of the entire output, each client's computation and communication complexity should be independent of the number of clients.

Next, we note that the each member of the honest majority committee D_i (a) reads the TFHE public key and ciphertexts sent by the key generation committee members, (b) reads the FHE output ciphertext, and (c) computes and outputs the partial decryption along with proof of correct decryption. This incurs a communication and computation complexity of $\text{poly}_\lambda(n, m)$.

Protocol Π_{Yoso}

Key Generation: Each key generation server K_i does the following:

1. Generate keys $(tpk_i, \{tsk_{i \rightarrow j}\}_{j \in [n]}) \leftarrow \text{TFHE.KGen}(\rho_{\text{KGen},i})$.
2. For $j \in [n]$, encrypt $ct_{i \rightarrow j} \leftarrow \text{Enc}(ppk_j, tsk_{i \rightarrow j}; \rho_{\text{KGen},i})$.
3. Let $\phi_{\text{KGen},i} = (ppk_1, \dots, ppk_n, tpk_i, ct_{i \rightarrow 1}, \dots, ct_{i \rightarrow n})$. Compute a proof $\pi_{\text{KGen},i} \leftarrow \text{Prove}(\text{crs}_{\text{KGen}}, \phi_{\text{KGen},i}, \rho_{\text{KGen},i})$.
4. Invoke $\mathcal{F}_{\text{PubComp}}$ with $(\text{input}, \text{inputwire} = (\text{KGen}, i), (tpk_i, \{ct_{i \rightarrow j}\}_{j \in [n]}, \pi_{\text{KGen},i}))$.

Input: Each client Client_i does the following:

1. Invoke $\mathcal{F}_{\text{PubComp}}$ with $(\text{output}, \text{KGen})$ to obtain the TFHE public key tpk .
2. Encrypt its input x_i under tpk to get $ct_i \leftarrow \text{TFHE.Enc}(tpk, x_i; \rho_{\text{Enc},i})$.
3. Let $\phi_{\text{Enc},i} = (tpk, ct_i)$.
4. Compute a zk-SNARK proof $\pi_{\text{Enc},i} \leftarrow \text{Prove}(\text{crs}_{\text{Enc}}, \phi_{\text{Enc},i}, (x_i, \rho_{\text{Enc},i}))$.
5. Invoke $\mathcal{F}_{\text{PubComp}}$ with $(\text{input}, \text{inputwire} = (\text{Eval}, i), (ct_i, \pi_{\text{Enc},i}))$.

Output Computation: Each member D_i of the honest-majority decryption committee does the following:

1. Invoke $\mathcal{F}_{\text{PubComp}}$ with $(\text{output}, \text{KGen})$ to obtain $(tpk, \{ct_{j \rightarrow i}\}_{j \in L_{\text{KGen}}})$.
2. Let $tsk_{j \rightarrow i} \leftarrow \text{Dec}(psk_i, ct_{j \rightarrow i})$ for $j \in L_{\text{KGen}}$.
3. Invoke $\mathcal{F}_{\text{PubComp}}$ with $(\text{output}, \text{Eval})$ to obtain ct .
4. Partially decrypt ct to obtain $d_i \leftarrow \text{TFHE.PDec}(tpk, \{tsk_{j \rightarrow i}\}_{j \in L_{\text{KGen}}}, ct)$.
5. Let $\phi_{\text{Dec},i} = (\{ct_{j \rightarrow i}\}_{j \in L_{\text{KGen}}}, ct_i, d_i, tpk)$.
6. Compute a zk-SNARK proof $\pi_{\text{Dec},i} \leftarrow \text{Prove}(\text{crs}_{\text{PDec}}, \phi_{\text{Dec},i}, psk_i)$.
7. Invoke $\mathcal{F}_{\text{PubComp}}$ with $(\text{input}, \text{inputwire} = (\text{Dec}, i), (d_i, \pi_{\text{Dec},i}))$.

Output: To get the output, a party P invokes $\mathcal{F}_{\text{PubComp}}$ with $(\text{output}, \text{Dec})$ to obtain y .

Fig. 10: Protocol Π_{Yoso}

5.5 Security Analysis

We state the formal theorem and prove the security of Π_{YOSO} below.

Theorem 4. *The protocol Π_{YOSO} YOSO-realizes the MPC functionality \mathcal{F}_{MPC} with guaranteed output delivery in the $\mathcal{F}_{\text{PubComp}}$ -hybrid model, as long as a majority of decryption servers is honest.*

Proof. To prove that protocol Π_{YOSO} YOSO realises the functionality \mathcal{F}_{MPC} with guaranteed output delivery, we will show that $\text{YoS}(\Pi_{\text{YOSO}})$ UC realises the functionality \mathcal{F}_{MPC} . We do this by constructing a simulator \mathcal{S} for any given adversary \mathcal{A} , such that

$$\text{REAL}_{\text{YoS}(\Pi_{\text{YOSO}}), \mathcal{A}, \mathcal{E}}(1^{1^\lambda}) \approx \text{IDEAL}_{\mathcal{F}_{\text{MPC}}, \mathcal{S}, \mathcal{E}}(1^{1^\lambda}).$$

For a given adversary \mathcal{A} , we define the simulator \mathcal{S} in Figure 11.

First, we note that the correctness of Π_{YOSO} follows from the correctness of the TFHE scheme, completeness of the zk-SNARK proof system and the correctness of the computations done via $\mathcal{F}_{\text{PubComp}}$. Next, we argue the indistinguishability of the real and ideal world via a series of hybrids.

Real H0: Run everything as in the real protocol, using the honest roles inputs.

Hybrid H1: In this hybrid, the aggregated key during key generation includes the key of corrupt K_i only if the randomness extracted via NIZK is consistent with $(tpk_i, \{ct_{i \rightarrow j}\}_{j \in [n]})$ sent by K_i . This is unlike the previous hybrid where the key of K_i is included as long as the zk-SNARK proof verifies. Indistinguishability follows due to the simulation extractability property (defined in 9) of the zk-SNARK used for key generation.

Hybrid H2: In this hybrid, the simulator queries the ideal functionality obtained by extracting the inputs of corrupt clients from the zk-SNARK proofs $\pi_{\text{Enc}, i}$ sent by Client_i to compute the output y . Indistinguishability follows due to the simulation extractability property (defined in 9) of the zk-SNARK used by the clients.

Hybrid H3: In this hybrid, the simulator switches the encryptions corresponding to honest parties to encrypt 0. Indistinguishability follows from semantic security of the TFHE scheme.

References

1. Acharya, A., Hazay, C., Kolesnikov, V., Prabhakaran, M.: SCALES - MPC with small clients and larger ephemeral servers. In: Kiltz, E., Vaikuntanathan, V. (eds.) TCC 2022, Part II. LNCS, vol. 13748, pp. 502–531. Springer, Cham, Switzerland, Chicago, IL, USA (Nov 7–10, 2022)

Simulator \mathcal{S} for Π_{Yoso}

Begin by obtaining the extraction trapdoor td_{KGen} and td_{Enc} , during the generation of crs_{KGen} , and crs_{Enc} . Allow the adversary to control the corrupt roles, simulating the honest roles and $\mathcal{F}_{\text{PubComp}}$. Let \mathcal{H} and \mathcal{I} denote the set of indices corresponding to honest and corrupt parties respectively.

Key Generation: Initialize a list L_{KGen} to keep track of verifying proofs.

1. Corresponding to each corrupt key generation server K_i , \mathcal{S} does the following upon receiving as input (**input**, **inputwire** = $(\text{KGen}, i), (tpk_i, \{ct_{i \rightarrow j}\}_{j \in [n]}, \pi_{\text{KGen}, i})$) on behalf of $\mathcal{F}_{\text{PubComp}}$.
 - Use td_{KGen} to extract $\rho_{\text{KGen}, i}$.
 - Recompute $(tpk_i, \{ct_{i \rightarrow j}\}_{j \in [n]})$ using $\rho_{\text{KGen}, i}$. If it is consistent with the version sent by K_i , add i to L_{KGen} .
2. On behalf of honest K_j ($j \in \mathcal{H}$), generate tpk_j and $\{ct_{j \rightarrow i}\}_{i \in [n]}$ as per the protocol specifications and add $j \in L_{\text{KGen}}$.
3. Aggregate the verifying TFHE public keys as $tpk \leftarrow \text{AggregateKeys}(\{tpk_i\}_{i \in L_{\text{KGen}}})$.
4. Store $(tpk, \{ct_{i \rightarrow j}\}_{i \in L_{\text{KGen}}, j \in [n]})$ as output of KGen.

Input: \mathcal{S} does the following:

1. When invoked by corrupt Client_i with (**output**, KGen), return the previously stored output tpk .
2. Corresponding to each corrupt client Client_i , \mathcal{S} does the following upon receiving as input (**input**, **inputwire** = $(\text{Eval}, i), (ct_i, \pi_{\text{Enc}, i})$) on behalf of $\mathcal{F}_{\text{PubComp}}$.
 - Use td_{Enc} to extract $(x_i, \rho_{\text{Enc}, i})$.
 - Recompute ct_i using tpk and $(x_i, \rho_{\text{Enc}, i})$. If it is inconsistent with the version sent by Client_i , replace it with an encryption of default input.
3. Compute $ct = \text{Eval}(tpk, C, ct_1, \dots, ct_\ell)$, where ct_j corresponding to $j \in \mathcal{H}$ are replaced with encryptions of 0.
4. Store ct as output of Eval.

At this stage, \mathcal{S} knows the inputs x_i for each corrupt client Client_i . The \mathcal{S} can provide this to the ideal functionality \mathcal{F}_{MPC} to obtain the output y .

Output Computation: \mathcal{S} does the following:

1. When invoked by corrupt D_i with (**output**, KGen), return the previously stored output $(tpk, \{ct_{j \rightarrow i}\}_{j \in L_{\text{KGen}}})$.
2. When invoked by corrupt D_i with (**output**, Eval), return the previously stored output ct .

Output : Return y on behalf of $\mathcal{F}_{\text{PubComp}}$ when invoked by any party with (**output**, Dec).

Fig. 11: Simulator for Π_{Yoso}

2. Acharya, A., Hazay, C., Kolesnikov, V., Prabhakaran, M.: Malicious security for SCALES: Outsourced computation with ephemeral servers. *Cryptology ePrint Archive, Report 2024/383* (2024), <https://eprint.iacr.org/2024/383>
3. Asharov, G., Jain, A., López-Alt, A., Tromer, E., Vaikuntanathan, V., Wichs, D.: Multiparty computation with low communication, computation and interaction via threshold FHE. In: Pointcheval, D., Johansson, T. (eds.) *EUROCRYPT 2012*. LNCS, vol. 7237, pp. 483–501. Springer, Berlin, Heidelberg, Germany (Apr 2012)
4. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. *Journal of Cryptology* 37(2), 18 (Apr 2024)
5. Benhamouda, F., Gentry, C., Gorbunov, S., Halevi, S., Krawczyk, H., Lin, C., Rabin, T., Reyzin, L.: Can a public blockchain keep a secret? In: Pass, R., Pietrzak, K. (eds.) *TCC 2020, Part I*. LNCS, vol. 12550, pp. 260–290. Springer, Cham, Switzerland (Nov 2020)
6. Bienstock, A., Escudero, D., Polychroniadou, A.: On linear communication complexity for (maximally) fluid MPC. In: Handschuh, H., Lysyanskaya, A. (eds.) *CRYPTO 2023, Part I*. LNCS, vol. 14081, pp. 263–294. Springer, Cham, Switzerland (Aug 20–24, 2023)
7. Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M.R., Sahai, A.: Threshold cryptosystems from threshold fully homomorphic encryption. In: Shacham, H., Boldyreva, A. (eds.) *CRYPTO 2018, Part I*. LNCS, vol. 10991, pp. 565–596. Springer, Cham, Switzerland (Aug 2018)
8. Braun, L., Damgård, I., Orlandi, C.: Secure multiparty computation from threshold encryption based on class groups. In: Handschuh, H., Lysyanskaya, A. (eds.) *CRYPTO 2023, Part I*. LNCS, vol. 14081, pp. 613–645. Springer, Cham, Switzerland (Aug 20–24, 2023)
9. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: *42nd FOCS*. pp. 136–145. IEEE Computer Society Press (Oct 2001)
10. Canetti, R., Kolby, S., Ravi, D., Soria-Vazquez, E., Yakoubov, S.: Taming adaptivity in YOSO protocols: The modular way. In: Rothblum, G.N., Wee, H. (eds.) *TCC 2023, Part II*. LNCS, vol. 14370, pp. 33–62. Springer, Cham, Switzerland, Taipei, Taiwan (Nov 29 – Dec 2, 2023)
11. Choudhuri, A.R., Goel, A., Green, M., Jain, A., Kaptchuk, G.: Fluid MPC: Secure multiparty computation with dynamic participants. In: Malkin, T., Peikert, C. (eds.) *CRYPTO 2021, Part II*. LNCS, vol. 12826, pp. 94–123. Springer, Cham, Switzerland, Virtual Event (Aug 16–20, 2021)
12. Choudhuri, A.R., Green, M., Jain, A., Kaptchuk, G., Miers, I.: Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) *ACM CCS 2017*. pp. 719–728. ACM Press (Oct / Nov 2017)
13. Cleve, R.: Limits on the security of coin flips when half the processors are faulty (extended abstract). In: *18th ACM STOC*. pp. 364–369. ACM Press (May 1986)
14. Deligios, G., Goel, A., Liu-Zhang, C.D.: Maximally-fluid MPC with guaranteed output delivery. *Cryptology ePrint Archive, Report 2023/415* (2023), <https://eprint.iacr.org/2023/415>
15. Ethereum: Optimistic rollups (2023), <https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/#multi-round-interactive-proving>
16. Ethereum: Zero-knowledge rollups (2023), <https://ethereum.org/en/developers/docs/scaling/zk-rollups/>
17. Fuchsbaauer, G., Kiltz, E., Loss, J.: The algebraic group model and its applications. In: Shacham, H., Boldyreva, A. (eds.) *Advances in Cryptology -*

- CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10992, pp. 33–62. Springer (2018), https://doi.org/10.1007/978-3-319-96881-0_2
18. Garg, S., Gentry, C., Sahai, A., Waters, B.: Witness encryption and its applications. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) 45th ACM STOC. pp. 467–476. ACM Press (Jun 2013)
 19. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 465–482. Springer, Berlin, Heidelberg, Germany (Aug 2010)
 20. Gentry, C., Halevi, S., Krawczyk, H., Magri, B., Nielsen, J.B., Rabin, T., Yakubov, S.: YOSO: You only speak once - secure MPC with stateless ephemeral roles. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part II. LNCS, vol. 12826, pp. 64–93. Springer, Cham, Switzerland, Virtual Event (Aug 16–20, 2021)
 21. Gentry, C., Halevi, S., Magri, B., Nielsen, J.B., Yakubov, S.: Random-index PIR and applications. In: Nissim, K., Waters, B. (eds.) TCC 2021, Part III. LNCS, vol. 13044, pp. 32–61. Springer, Cham, Switzerland, Raleigh, NC, USA (Nov 8–11, 2021)
 22. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 75–92. Springer, Berlin, Heidelberg, Germany (Aug 2013)
 23. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th ACM STOC. pp. 218–229. ACM Press (May 1987)
 24. Gordon, S.D., Liu, F.H., Shi, E.: Constant-round MPC with fairness and guarantee of output delivery. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part II. LNCS, vol. 9216, pp. 63–82. Springer, Berlin, Heidelberg, Germany (Aug 2015)
 25. Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part II. LNCS, vol. 9666, pp. 305–326. Springer, Berlin, Heidelberg, Germany (May 2016)
 26. Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J. (eds.) EUROCRYPT. Lecture Notes in Computer Science, vol. 9666, pp. 305–326. Springer (2016)
 27. Kahn, A.B.: Topological sorting of large networks. *Communications of the ACM* 5(11), 558–562 (1962)
 28. Kolby, S., Ravi, D., Yakubov, S.: Towards efficient YOSO MPC without setup. *Cryptology ePrint Archive, Report 2022/187* (2022), <https://eprint.iacr.org/2022/187>
 29. Kosba, A.E., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE Symposium on Security and Privacy. pp. 839–858. IEEE Computer Society Press (May 2016)
 30. Kosba, A.E., Zhao, Z., Miller, A., Qian, Y., Chan, T.H., Papamanthou, C., Pass, R., Shelat, A., Shi, E.: How to use snarks in universally composable protocols. *IACR Cryptol. ePrint Arch.* p. 1093 (2015), <http://eprint.iacr.org/2015/1093>
 31. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO’89. LNCS, vol. 435, pp. 218–238. Springer, New York, USA (Aug 1990)
 32. Rachuri, R., Scholl, P.: Le mans: Dynamic and fluid MPC for dishonest majority. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part I. LNCS, vol. 13507, pp. 719–749. Springer, Cham, Switzerland (Aug 15–18, 2022)

33. Rivinius, M., Reisert, P., Rausch, D., Küsters, R.: Publicly accountable robust multi-party computation. In: 2022 IEEE Symposium on Security and Privacy. pp. 2430–2449. IEEE Computer Society Press, San Francisco, CA, USA (May 22–26, 2022)
34. Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: 27th FOCS. pp. 162–167. IEEE Computer Society Press (Oct 1986)

A Building Blocks

In this section, we describe the building blocks used by our constructions. In Section A.1 we recall threshold fully homomorphic encryption (TFHE); in Section A.2 describe the zero knowledge tools we use. Our constructions also rely on Merkle trees [31] and collision-resistant hashing, which we assume our readers are familiar with.

A.1 Threshold Fully Homomorphic Encryption

In this section, we go over the definition of threshold fully homomorphic encryption (TFHE). We use the definitions and instantiation from Kolby *et al* [28], which adapts the GSW-style [22] TFHE scheme from Gordon *et al* [24].

Syntax Kolby *et al* define a threshold key generation and decryption interface permitting non-interactive key aggregation and decryption of ciphertexts. In the non-interactive key generation phase, each key generation server sends both a public key contribution and a *sharing* of the corresponding secret key contribution. Homomorphic evaluation of the desired function circuit can be performed over ciphertexts encrypted to the same (aggregated) set of joint keys. Decryption servers holding shares of the joint public key produce partial decryption shares d , which are again aggregated non-interactively.

Kolby *et al* describe the following TFHE syntax. We augment their syntax by adding the **AggregateKeys** algorithm (in their scheme, it is instantiated simply by adding the public keys together), and by removing the circuit depth from the syntax altogether (that dependency can be lifted through bootstrapping).

Setup($1^\lambda, n; \rho$) \rightarrow **pp**: A setup algorithm parameterized by the size n of the honest-majority decryption committee, producing public parameters **pp**, which are given as an implicit argument to all subsequent algorithms.

KGen(ρ_i) \rightarrow (tpk_i, tsk_i): Given public parameters **pp** and randomness ρ_i , the key generation algorithm produces a public key tpk_i and a secret key tsk_i split into shares, such that $tsk_i = (tsk_{i,1}, \dots, tsk_{i,n})$.

AggregateKeys($\{tpk_i\}_{i \in \mathcal{K}}$) \rightarrow tpk : Given a set of public keys, the aggregation algorithm produces a single public key.

Enc($tpk, x; \rho$) \rightarrow c : Given a public key tpk and a message x , the encryption algorithm encrypts to a ciphertext c under randomness ρ .

Eval(C, c_1, \dots, c_ℓ) \rightarrow c : Homomorphically evaluates circuit C on input ciphertexts c_1, \dots, c_ℓ to produce c .

$\text{PDec}(tpk, tsk_j, c) \rightarrow d_j$: For a ciphertext c , encrypted under the public keys $\{tpk_i\}_{i \in \mathcal{K}}$, and decryption secret key $csk_j = \{tsk_{i,j}\}_{i \in \mathcal{K}}$ this algorithm produces a partial decryption d_j .

$\text{Combine}(tpk, c, \{d_i\}_{i \in R}) \rightarrow y$: Given a set of partial decryptions $\{d_i\}_{i \in R}$ of size at least $t + 1$, decrypts ciphertext c to obtain plaintext y .

Properties A TFHE scheme should be *correct* and *decryption share simulatable* as defined by Kolby *et al.* It should also be *semantically secure* (Definition 2). (We re-state this definition because we use it explicitly in our proofs.)

Definition 2 (Semantic Security [28]). A TFHE scheme is *semantically secure under chosen plaintext attack* if, for all PPT adversaries \mathcal{A} ,

$$\text{Adv}^{\text{IND-CPA}} = \Pr[\mathcal{A} \text{ wins } \text{Game}_{\text{TFHE}}^{\text{IND-CPA}}] - \frac{1}{2} \leq \epsilon$$

for a negligible function ϵ in the security parameter 1^λ . Where $\text{Game}_{\text{TFHE}}^{\text{IND-CPA}}$ is defined as described in Fig. 12 and Fig.13.

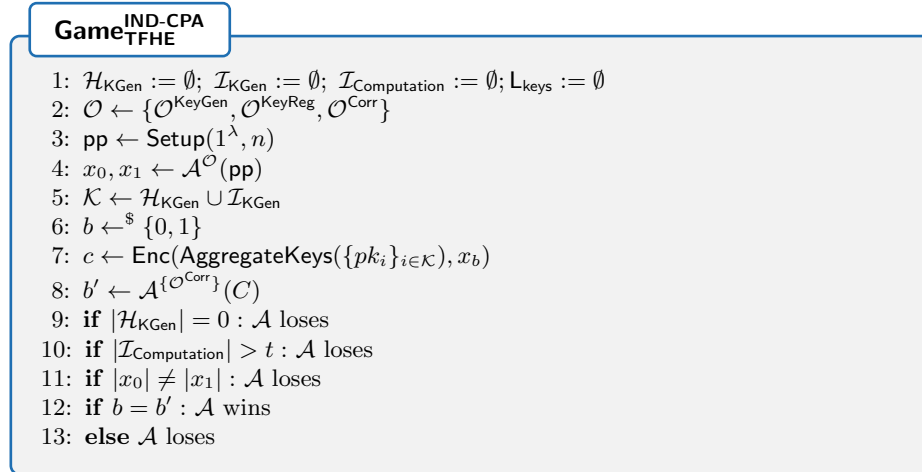
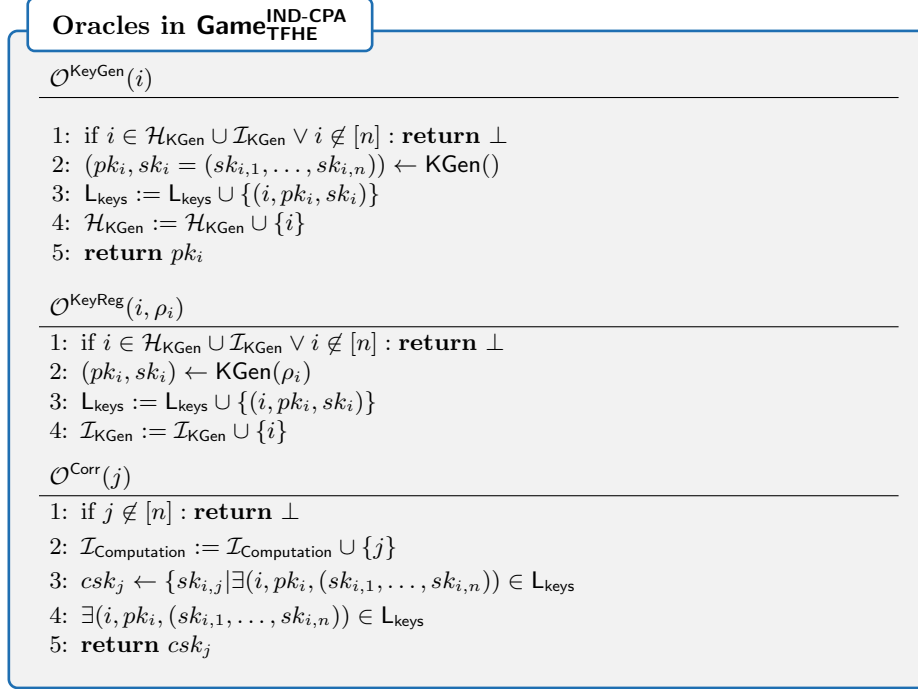


Fig. 12: Semantic security realized by the TFHE scheme of [28].

Realization The TFHE scheme from [28] realizes a notion of semantic security, reproduced in Fig. 12, which allows the adversary to choose *all-but-one key* contributions during the key generation phase. This is a critical, as we only afford a single honest majority decryption committee, but must assume dishonest majority for all other phases including key generation.

Fig. 13: Oracles used in the Game^{IND-CPA}_{TFHE}.

A.2 Argument Systems and Zero Knowledge

We reproduce definitions of Succinct Non-interactive Arguments (SNARG) and Succinct Non-Interactive Arguments of Knowledge (SNARK) from [26,30] and standard properties such as soundness, zero-knowledge and simulation extractability.

Definition 3 (SNARG). Let $R(\phi, \omega)$ be an NP relation corresponding to an NP language L . A Succinct Non-interactive Argument (SNARG) for relation R is a tuple of algorithms (Setup, Prove, Verify) defined as follows:

- $(\sigma, \tau) \leftarrow \text{Setup}(1^\lambda)$: The Setup takes the security parameter λ and outputs a common reference string (crs) σ and a simulation trapdoor τ for the relation R .
- $\pi \leftarrow \text{Prove}(\sigma, \phi, \omega)$: The prove algorithm takes as input the crs σ and the statement ϕ and witness ω such that $(x, \omega) \in R$ and returns an argument π .
- $0/1 \leftarrow \text{Verify}(\sigma, \phi, \pi)$: The verify algorithm takes as input the crs σ , the statement ϕ and an argument π and outputs 0 or 1 corresponding to reject and accept respectively.

Definition 4 (zk-SNARK). Let $R(\phi, \omega)$ be an NP relation corresponding to an NP language L . A Zero-Knowledge Non-interactive Argument of Knowledge

(zk-SNARK) for relation R is a tuple of algorithms (Setup, Prove, Verify, Sim) which has the same syntax as SNARG (defined in Definition 3) but with the additional algorithm Sim defined as follows:

- $\pi \leftarrow \text{Sim}(\sigma, \tau, \phi)$: The simulation takes as input a crs σ , a simulation trapdoor τ and a statement ϕ and returns an argument π .

Definition 5 (Succinctness). The proof string π is sublinear or polylogarithmic in the statement ϕ and ω .

Definition 6 (Completeness). For all $\lambda \in \mathbb{N}$ and $\phi \in L$:

$$\Pr \left[(\sigma, \tau) \leftarrow \text{Setup}(1^\lambda); \pi \leftarrow \text{Prove}(\sigma, \phi, \omega) : \text{Verify}(\sigma, \phi, \pi) = 1 \right] = 1$$

Definition 7 (Soundness). A non-interactive argument scheme is sound if for all non-uniform PPT adversary \mathcal{A} , $\lambda \in \mathbb{N}$ and given auxiliary information z , there exist a negligible function $\epsilon(\cdot)$:

$$\Pr \left[(\sigma, \tau) \leftarrow \text{Setup}(1^\lambda); (\phi, \pi) \leftarrow \mathcal{A}(\sigma, z) : \text{Verify}(\sigma, \phi, \pi) = 1 \wedge \phi \notin L_R \right] \leq \epsilon(\lambda)$$

SNARG and SNARK schemes which also satisfy zero-knowledge are zk-SNARGs and zkSNARKs respectively.

Definition 8 (Zero-Knowledge). For all $\lambda \in \mathbb{N}$ and $(\phi, \omega) \in R$, auxiliary information z and every PPT adversary \mathcal{A} , there exist a negligible function ϵ such that

$$\left| \Pr \left[(\sigma, \tau) \leftarrow \text{Setup}(1^\lambda); \pi \leftarrow \text{Prove}(\sigma, \phi, \omega) : \mathcal{A}(\sigma, \phi, \pi, \tau) = 1 \right] - \Pr \left[(\sigma, \tau) \leftarrow \text{Setup}(1^\lambda); \pi \leftarrow \text{Sim}(\sigma, \phi, \omega) : \mathcal{A}(\sigma, \phi, \pi, \tau) = 1 \right] \right| \leq \epsilon(\lambda)$$

In this work, our protocols can be instantiated with a SNARG (for public computation) and zk-SNARKs with simulation-extraction (for proving well-formedness of encrypted messages). Simulation extractability is required by the UC simulator of Π_{YOSO} in Fig.11.

Definition 9 (Strong Simulation-Extractability). A non-interactive zero-knowledge argument scheme (Setup, Prove, Verify, Sim) is Strong Simulation Extractable if there exist an extractor \mathcal{E} and a negligible function ϵ such that for any PPT adversary \mathcal{A} and R :

$$\Pr \left[(\sigma, \tau) \leftarrow \text{Setup}(1^\lambda); (\pi, \phi) \leftarrow \mathcal{A}^{S_{\sigma, \tau}}(\sigma); w \leftarrow \mathcal{E}(\sigma, \tau, \phi, \pi) : \text{Verify}(\sigma, \phi, \pi) = 1 \wedge (\phi, \omega) \notin R \wedge (\pi, \phi) \notin Q \right] \leq \epsilon(\lambda)$$

where $S_{\sigma, \tau}(\phi)$ is a simulation oracle that runs $\text{Sim}(\sigma, \tau, \phi)$ internally and also records $(\pi, \phi) \in Q$.

Realization In our work, we assume a zkSNARK scheme such as Groth16 [26] in the complexity analysis of our protocols. Groth 16 offers $O_\lambda(1)$ proof size (succinctness), $O_\lambda(|\ell|)$ verification cost (sublinear in the size of the computation) and proving time of $O_\lambda(|C| \log |C|)$; here, λ is the security parameter, $|\ell|$ is the size of the inputs, $|C|$ is the size of the circuit and $O_\lambda(\cdot)$ hides a multiplicative factor of $\text{poly}(\lambda)$.

In [17], they show that Groth16 is sound under the (q_1, q_2) -dlog assumption in the Algebraic Group Model (AGM). To satisfy the extra property of Strong Simulation-Extractability (defined in Definition 9), our protocol can be instantiated with the Groth16 variant proposed in [30]. This will affect the complexity with almost $\text{poly}(\lambda)$ factor, maintaining overall the complexities.