OLYMPIA: A Simulation Framework for Evaluating the Concrete Scalability of Secure Aggregation Protocols

Ivoline C. Ngong
University of Vermont
kngongiv@uvm.edu

Nicholas Gibson
University of Vermont
Nicholas.Gibson@uvm.edu

Joseph P. Near
University of Vermont
jnear@uvm.edu

***Abstract*—Recent secure aggregation protocols enable privacy-preserving federated learning for high-dimensional models among thousands or even millions of participants. Due to the scale of these use cases, however, end-to-end empirical evaluation of these protocols is impossible. We present OLYMPIA, a framework for empirical evaluation of secure protocols via simulation. OLYMPIA provides an embedded domain-specific language for defining protocols, and a simulation framework for evaluating their performance. We implement several recent secure aggregation protocols using OLYMPIA, and perform the first empirical comparison of their end-to-end running times. We release OLYMPIA as open source.**

I. INTRODUCTION

Federated learning [20] allows for collaborative distributed training of machine learning models without requiring training data to be collected centrally. By keeping training data decentralized, federated learning can reduce privacy risks for individuals who contribute training data. However, recent work has shown that in some cases, the individual model updates computed during federated learning can reveal a surprising amount about the original training data.

To address this challenge, *secure aggregation* protocols can be used to construct federated learning systems that reveal only *aggregated* model updates, providing much stronger protection against privacy attacks. Combined with differential privacy [13], secure aggregation protocols can enable truly privacy-preserving federated learning. Recent work in secure aggregation has produced protocols that scale to high-dimensional model updates [5] and millions of clients [3]; in theory, these approaches scale well enough to meet the requirements of industry-scale machine learning.

However, evaluating these protocols empirically remains a major challenge, because of the sheer scale of the use cases they are designed for. For example, evaluating a secure aggregation protocol with 10,000 clients is impossible for most researchers, because it requires provisioning 10,000 physical machines to perform the experiment. As a result, previous work has focused on evaluating individual components of a protocol in isolation [5] or simply reporting properties of the protocol's *expected* performance by analyzing the protocol itself [3]. Such evaluations are effective for comparing the asymptotic complexities of protocols, but may not capture the protocol's concrete performance.

We present OLYMPIA, a simulation framework for the empirical evaluation of secure aggregation protocols. OLYMPIA is designed to evaluate the concrete, *end-to-end* performance of protocols *at scale*, by leveraging an accurate simulation of hundreds or thousands of parties on a single machine. For example, OLYMPIA can perform a simulation of the Bell et al. [3] protocol for 10,000 clients on a single machine in just a few hours.

The OLYMPIA framework provides a simulator that accurately measures the end-to-end running time of protocols, including both communication and computation time. To model computation cost in many-party protocols, OLYMPIA records the actual computation time for each party, and simulates these computations running in parallel. To model communication cost, OLYMPIA uses a model of network latency based on actual internet latency data collected from internet speed tests. OLYMPIA builds on the existing ABIDES framework [6] to coordinate the simultaneous execution and communication of the parties and measure the total running time of the protocol.

To ease the implementation of new protocols, OLYMPIA provides a domain-specific language (DSL) embedded in Python for defining synchronous secure aggregation protocols. The OLYMPIA DSL makes it straightforward to translate protocol descriptions into implementations, and also provides utilities for common cryptographic constructs like public-key encryption and secret sharing. We have used the OLYMPIA DSL to implement existing several protocols from the literature in fewer than 100 lines of code.

We use OLYMPIA to conduct an empirical comparison between several existing protocols implemented in our case studies. The results are mostly consistent with existing conclusions about protocol performance, but also yield new insights about the concrete performance of these protocols. For example, we show that network latency has relatively little effect on the total running time for these protocols, and that packed secret sharing has a significant impact on performance for some protocols. In addition, we empirically validate the simulator's accuracy by comparing it against "ground truth" performance results obtained by executing the same protocols on real hardware.

We release the OLYMPIA framework and our case study implementations as open source.[1] In addition to comparing the performance of existing protocols, we hope that OLYMPIA

[1] https://github.com/uvm-plaid/olympia

will be useful as a standardized benchmarking tool for new protocols, and also as a tool for helping to refine existing protocols for better performance, to develop new protocols, and to evaluate protocol suitability for specific real-world deployment scenarios.

Contributions. In summary, our contributions are:

- We present OLYMPIA, a simulation framework for secure aggregation protocols that accurately models the concrete performance of protocols with millions of participants
- We use OLYMPIA to evaluate several existing protocols at scales that are not practical without a simulator
- We show that evaluation with OLYMPIA leads to important insights about the concrete performance of protocols, including some that can help improve protocol performance
- We validate OLYMPIA's simulation accuracy by comparing its results against actual execution times for a small number of clients

II. OVERVIEW OF OLYMPIA

Traditional *secure multiparty computation* (MPC) [14] protocols are designed to work best for a handful of parties—2- and 3-party computation are most common, and most protocols are evaluated with a single-digit number of parties. Some more recent protocols have been evaluated using as many as 128 parties [34]. At this scale, empirical evaluation is possible: a separate physical machine can be used for each party, allowing realistic measurement of total running time.

Secure aggregation protocols are designed to work for much larger sets of parties—typically, hundreds to thousands (or even millions). At this scale, experimental evaluation is not practical; it's simply not feasible to provision enough machines. Instead, authors typically report computation and communication *complexity* as a proxy for experimental results [3]; in some cases, authors additionally implement the protocol and measure concrete computation time for a *single* client [5] and for a server with fixed client inputs.

For secure aggregation protocols, authors typically **do not report end-to-end wall-clock time** resulting from an experimental evaluation, because it is not feasible to run such an experiment [5], [3], [30], [19], [32], [35], [17], [24], [25]. As we discuss later, the inability to measure concrete performance of these protocols makes it difficult to understand their relative performance properties.

OLYMPIA: Evaluation via Simulation. The goal of this work is to enable experimental evaluation of secure aggregation protocols *at scale*. Experimental evaluation with OLYMPIA can highlight surprising mismatches between analytic bounds and concrete performance and can suggest simple methods for significantly improving protocol performance. In addition, OLYMPIA can simplify deployment, by enabling developers to predict performance in advance.

Challenge: realism & scale. The primary challenge lies in building a framework that works at scale and produces accurate results. OLYMPIA is built on ABIDES [6]—a simulation framework originally designed for high-frequency trading applications in financial markets. ABIDES is designed to simulate a large number of *agents* running simultaneously and communicating asynchronously. ABIDES simulates concurrent execution of these agents with high precision and accuracy. ABIDES has previously been used to evaluate the performance of specific secure aggregation protocols [7], [17], [8]. To support the evaluation of secure aggregation protocols, OLYMPIA adds cost models for network traffic (based on real-world latency data and configurable bandwidth limitations) and computation (based on actual execution time), and a DSL that simplifies the specification of new protocols.

New Insights from OLYMPIA. By enabling empirical analysis at scale, OLYMPIA can lead to important insights about concrete protocol performance. We implemented several state-of-the-art secure aggregation protocols in OLYMPIA (Section IV); our experimental evaluation of these protocols (Section V) confirms their expected asymptotic behavior, but also surfaces performance properties not obvious from the asymptotics alone. For example, the results suggest that computation time has a much larger effect on total running time than network latency or round complexity; in addition, for one protocol, our results demonstrate the practical importance of optimizations like packed secret sharing that improve concrete performance without changing computation or communication complexity.

III. THE OLYMPIA FRAMEWORK

OLYMPIA provides two main components: a domain-specific language (DSL) for describing single-server secure aggregation protocols, and a simulation framework for evaluating the practical concrete performance of these protocols.

The OLYMPIA DSL. As a DSL, this framework could potentially be used for extracting a protocol's implementation, evaluating its performance based on round complexity, computational complexity, bandwidth cost, and hyper parameter impact, etc. We describe the OLYMPIA DSL in Section III-A.

The OLYMPIA Simulator. Its implementation is based on the discrete event simulation framework ABIDES, but implements the round-based synchronous communication commonly used in secure aggregation protocols. In addition, the OLYMPIA simulator is specifically designed to accurately record both computation and communication cost, by using realistic models for both components. We describe the OLYMPIA simulator in Section III-B.

A. The OLYMPIA DSL

Protocols in OLYMPIA are defined in three parts: a *server class* implementing the aggregation server's behavior, a *client class* implementing the client's behavior, and a *config file* that determines protocol setup and parameters. Multiple experiments can be quickly reconfigured and run with varying simulation parameters using a single setup.

The server and client classes implement what each party does in each round. Clients and servers are defined in OLYMPIA by inheriting from the `AggregationClient` and `AggregationServer` classes, which implement the minimal set of properties and methods necessary for efficient communication between agents and interaction with simulation kernels. Client and server classes override methods to define the behavior of the protocol. Table I describes the OLYMPIA API in terms of the key methods used in defining a protocol.

Method	Description
`round()`: *round #* × *message(s)* → *next message(s)* **(server and client)**	The server is given the round number and incoming messages from the agents and implements what takes place in each round.
`next_round()`: *round #* × *message(s)* → *bool* **(server only)**	Given the current round number and incoming messages from the clients, determines whether it is time to move to the next round.
`succeed()` **(server only)**	Called at the end of the protocol, to indicate success and record results
`fail()` **(server only)**	Called during the protocol to indicate failure

TABLE I: OLYMPIA API (server and clients).

```
class BaselineClient(AggregationClient):
  def round(self, round_number, message):
    if round_number == 1:   # send secret input to server
      return self.GF(self.secret_input)

class BaselineServer(DropoutAggregationServer):
  def round(self, round_number, messages):
    if round_number == 1:   # start the protocol
      self.threshold = int(len(self.clients) * .95)
      return {client: None for client in self.clients}

    elif round_number == 2: # sum up received vectors
      self.succeed(GF(list(messages.values())).sum(axis=0))
```

Fig. 1: OLYMPIA implementation of the baseline protocol.

Example: Baseline Insecure Summation Protocol. Here, we present a simple example that allows clients to submit their inputs to a *trusted* server, which computes the sum of these inputs. Though this protocol is *insecure*, it serves as a good example for how protocols are defined in OLYMPIA, and a useful baseline for performance comparisons in our evaluation.

After setting up all required parameters in the baseline configuration file, the protocol is implemented as shown in Figure 1. The server for this protocol (and for the other protocols we implement later) inherit from the `DropoutAggregationServer`, which simulates a fraction of clients dropping out during each round of the protocol.

B. *The* OLYMPIA *Simulator*

The OLYMPIA simulator is based on ABIDES, a Python application built around actors called agents, who communicate via *asynchronous* messages. In the ABIDES framework, agents are instances of subclasses that inherit from the base `Agent` class. Subclasses implement what each agent does in a protocol, with the same type of agents having only one subclass. For instance, it is sufficient to create two subclasses for a protocol with a single server and multiple clients, although each client can have different attributes like timing. The base Agent class in ABIDES implements the minimum required methods for each agent to properly interact with the simulation kernel.

ABIDES lacks important functionalities for our setting, which OLYMPIA's simulator implements: measuring communication costs and computation time, modeling a WAN network, simulating dropouts realistically, and evaluating how other significant parameters (input vector size, finite field size, etc.) affect the protocol.

Overview of simulation approach. The OLYMPIA simulator runs the protocol by executing its components sequentially, passing messages between the parties as the protocol specifies. The simulator measures the execution time of each component individually, and calculates total running time by *simulating* the parallelism of actual protocol execution. When two component executions could run in parallel (because neither depends on a message from the other—e.g. when two clients compute their responses to a message from the server), then the simulator calculates the total time of both components as the maximum of their execution times (rather than the sum). This approach allows simulating protocols with thousands of parties on a single computer, while accurately modeling the parallelism that would occur in actual execution of the protocol.

Modeling computation time. OLYMPIA models computation delays by measuring the actual time the protocol takes to complete the computation, and accounting for that time as computation time spent by the appropriate party in the protocol. In ABIDES, computation time is measured by asking the programmer to specific explicit *computation delays*, and tracking each party's individual time using these delays as the simulation progresses.

In OLYMPIA, on the other hand, the simulator measures actual computation time for each party. The simulation runs each party's computation sequentially, and then aggregates the computation delays to simulate the parallel execution of client computations. While this enables us to capture realistic computation times, it also means that accurate results depend on the actual efficiency of the protocol's implementation. In our case study implementations, we use efficient libraries to implement computationally challenging features (e.g. PyNaCl for public-key encryption; Galois for finite field operations).

Modeling network latency. ABIDES supports the modeling of communication latency between different parties in the protocol, which is crucial to creating realistic simulations. In this model, a two-dimensional *latency matrix* defines the minimum nanosecond delay between each pair of parties. The kernel uses this matrix and a noise model to simulate network conditions.

OLYMPIA models network delays using a real-world internet speed test dataset [22] to simulate deployment on a wide-area network. Based on zoom level 16 web mercator tiles (approximately 610.8 meters by 610.8 meters at the equator), the dataset provides global fixed broadband and mobile (cellular) network latency measurements. The latency matrix is computed by measuring the latency between each endpoint and their respective internet providers, then adding the speed-of-light latency between their geographic locations.

Modeling network bandwidth limitations. In addition to modeling network latency, OLYMPIA also provides added support for modeling network bandwidth limitations. Bandwidth limits can be configured for both the clients and the server, and are incorporated into the total running time measured by the simulator. The default setting in OLYMPIA is no bandwidth limit; adjusting the limit can be especially useful to enable modeling of situations where the server has limited bandwidth, as described in our evaluation.

Modeling dropouts. Many secure aggregation protocols (in-

cluding the case studies described in Section IV) are robust to a fraction of the clients dropping out during the execution of the protocol. In some cases, clients dropping out can affect protocol performance, so OLYMPIA is capable of simulating dropouts in order to surface these effects. `DropoutAggregationServer` represents a server intended to be robust against dropouts; implementations specify an upper bound δ on the expected fraction of dropouts, and the server moves on to the next round after it has received at least $n\delta$ messages from clients.

Checking correctness. OLYMPIA is not designed to verify the correctness of protocol implementations, but it can be used as a tool for end-to-end testing of protocols. Our case studies are implemented to facilitate this kind of testing, by providing consistent inputs on each run of the protocol and checking that the protocol's output is as expected. OLYMPIA's simulator can be used to test correctness under varying conditions that may surface bugs—for example, we found and fixed several bugs in our implementations by varying the number of clients and the simulated dropout rate.

Threat model. Our case study protocols can be implemented with either semi-honest or malicious security. OLYMPIA does not actually simulate malicious parties, and is not designed to evaluate security. However, OLYMPIA can be used to evaluate the additional overhead of malicious-secure protocol variants. For our case studies, we implemented both semi-honest and malicious-secure variants, and our evaluation includes a comparison of the associated overhead.

Libraries and cryptographic primitives. OLYMPIA is primarily Python-based, and integrates most easily with other Python libraries. To produce high-performance, practically secure protocol implementations, most protocol implementations will leverage Python wrappers around efficient, well-tested libraries (usually implemented in C). For example, our case studies use PyNaCl (a wrapper around libsodium) and Galois (a finite field library that uses BLAS/LAPACK for array operations). Usage of well-tested libraries for cryptographic building blocks can also prevent subtle side-channel vulnerabilities.

Configuration and experiments. Experiments in OLYMPIA are configured using a YAML file that specifies the parameters to vary (e.g. the number of clients, the dimensionality of the inputs, and other protocol parameters). Given a configuration file, OLYMPIA runs the specified experiments on a single machine and saves the results to a CSV file. OLYMPIA is capable of running experiments involving tens of thousands of clients and vectors with millions of elements on a single machine (our experiments required only 64GB of memory).

C. Supported Protocol Designs

OLYMPIA is designed to support a variety of protocol types and architectures—the only restriction is that the protocol must be synchronous and proceed in rounds. Protocol implementations can contain arbitrary Python code, and messages between parties can contain any Python value whose size in bytes can be measured. Though our case studies focus on single-server secure aggregation protocols, OLYMPIA's flexibility makes it suitable for evaluating many other protocols as well.

Single-server secure aggregation. Our case studies focus on single-server secure aggregation protocols, because protocols in this category are often designed for hundreds or thousands of parties and are therefore especially difficult to evaluate by other means. OLYMPIA provides API support for this setting, by defining `AggregationServer` and `AggregationClient` classes to be extended by protocol implementations.

Protocols involving multiple servers. OLYMPIA can also support multi-server protocols like Prio [9] and its descendants. Implementing multi-server protocols requires defining a subset of the parties to be servers, and defining a class that specifies the servers' behavior, and then defining the clients to send messages directly to the multiple servers. This setting requires clients to send messages to multiple servers; to implement it in OLYMPIA, the programmer would need to implement new client and server classes with explicit `send` operations to specify the communication pattern.

Peer-to-peer protocols. OLYMPIA also supports peer-to-peer protocols, including general MPC protocols that evaluate circuits. These protocols are not typically designed to scale to thousands of parties, and can often be evaluated empirically without OLYMPIA, so our case studies do not focus on protocols of this type. Recent work designed to scale to larger numbers of parties [16] may benefit from evaluation using OLYMPIA. This setting requires clients to send messages to each other; like the multi-server setting, the programmer would need to specify each `send` operation manually to implement it.

Input validation. Recent approaches for secure aggregation with input validation [9], [26], [2] can also be evaluated using OLYMPIA. These approaches typically add a layer on top of an existing protocol, and do not change the protocol's communication patterns; as long as the additional layer can be implemented using Python, it can be evaluated in OLYMPIA.

Threat models. OLYMPIA does not evaluate security, and it is agnostic to the protocol's intended threat model. Variants of protocols that target different threat models (including all possible combinations of corrupted and honest parties) can be implemented and compared in OLYMPIA. Our evaluation includes semi-honest and malicious-secure variants of several protocols.

Limitations. OLYMPIA is designed mainly for single-server, synchronous protocols and although it can support any synchronous protocol, its use is limited in asynchronous settings. Since it is based on Python, integrating code from other programming languages can be challenging, limiting its flexibility. Additionally, the fact that computations are executed sequentially for each client also means that experiments can be time-consuming, particularly for more complex protocols. Moreover, implementing peer-to-peer protocols demands extra coding effort, as OLYMPIA's DSL is primarily oriented towards single-server and client-server protocols.

IV. CASE STUDIES

This section describes our implementations of six secure aggregation protocols in OLYMPIA. The simplest of these uses secret sharing to add the input vectors (§ IV-B); it requires $O(nl)$ per-client communication for n clients and vectors of length l, so it is not practical for large n or large l. The

protocols of Stevens et al. [32] (§ IV-C) and Bonawitz et al. [5] (§ IV-D) improve per-client communication cost to $O(n+l)$; these protocols work well for large vectors, but not for huge sets of clients. Finally, the Bell et al. [3] (§ IV-E), Sharing sharing [31], and ACORN [2] protocols improve per-client communication cost even further—to $O(\log n + l)$—making them suitable for large sets of clients. Table II summarizes our case studies.

A. Common Elements

This section describes some common elements used in all of the protocols implemented in our case studies.

Principle: peer-to-peer communication via the server. The OLYMPIA API is designed for building single-server aggregation protocols, in which clients communicate only with the server. This setup is designed to model real-world constraints on federated learning systems, in which clients are often mobile devices that may be protected by a firewall, and clients may have limited connectivity. However, many secure aggregation protocols require clients to send messages to each other, and require hiding the contents of those messages from the server. To accomplish this, single-server secure aggregation protocols typically use public-key cryptography to allow the clients to communicate *through* the server, and the server simply routes messages to the correct clients. All of the protocols described in this section make use of this approach.

Building block: public-key encryption. We make the same assumptions about public-key cryptography as Bonawitz et al. [5]. Specifically, the protocols described in this section rely on Diffie-Hellman key agreement [12], which describes a *key generation* procedure to generate public and private keys, and a *key agreement* procedure that combines party a's private key with party b's public key to form a shared symmetric encryption key—and produces the *same* symmetric key when performed in reverse (i.e. $\textbf{agree}(sk_a, pk_b) = \textbf{agree}(sk_b, pk_a)$). In our implementations, we implement public-key cryptography using PyNaCl, a wrapper around the efficient libsodium library.

Building block: threshold secret sharing. A (t, n) *threshold secret sharing* scheme allows a party to split a secret input into n *shares*, such that each individual share reveals nothing about the secret input, but t shares enable reconstruction of the secret. Such schemes also have an *additive homomorphism*—they allow adding shares of *different* secrets together to compute one share of the sum of the secrets. The most commonly-used example is Shamir's secret sharing scheme [29].

OLYMPIA provides an efficient implementation of Shamir secret sharing, leveraging the Galois library for working with finite fields. We also provide an implementation of *packed secret sharing*, which encodes more than one secret in a single share. This scheme adds a parameter k to the `share` function; it encodes k field elements in a single share, but requires at least $t + k$ shares for reconstruction. Packed secret sharing can improve concrete performance in some protocols, as our evaluation shows.

B. Secret Sharing Protocol

Our first case study is a simple protocol that uses threshold secret sharing to perform secure aggregation. This protocol was designed as a simple secure baseline, and only performs well when both the number of clients (n) and the size of the aggregated vectors (l) are small.

```
1  class SecretSharingClient(AggregationClient):
2    def round(self, round_number, message):
3      if round_number == 1:    # generate keys
4        self.sk_u = PrivateKey.generate()
5        return self.sk_u.public_key
6      elif round_number == 2: # generate shares
7        self.pks, n = message, len(message)
8        shares = shamir.share_array(self.secret_input,
9                                    n, n//2)
10       return {c: shares[c].encrypt(self.sk_u, pk)
11               for c, pk in self.pks.items()}
12     elif round_number == 3: # sum up received shares
13       dec_shares = [s.decrypt(self.sk_u, self.pks[c])
14                     for c, s in message.items()]
15       return shamir.sum_share_array(dec_shares, axis=0)
```

Fig. 2: OLYMPIA implementation of the secret sharing protocol (client).

```
1  class SecretSharingServer(DropoutAggregationServer):
2    def round(self, round_number, messages):
3      if round_number == 1:    # start the protocol
4        return {client: None for client in self.clients}
5      elif round_number == 2: # broadcast public keys
6        return {client: messages for client in self.clients}
7      elif round_number == 3: # route shares to clients
8        return route_messages(messages)
9      elif round_number == 4: # reconstruct sum
10       vs = messages.values()
11       self.succeed(shamir.reconstruct_array(vs))
```

Fig. 3: OLYMPIA implementation of the secret sharing protocol (server).

Protocol description. For n clients and one server, aggregating vectors of field elements with length l, the high-level idea of the protocol is as follows: first, each client sends one share of its input to each other client; second, each client sums the shares it receives to compute one share of the total sum, and sends this share to the server; third, the server uses the shares to reconstruct the sum. Since each client needs to generate n shares for each of l vector elements, the per-client communication cost is $O(nl)$.

Protocol implementation. The complete OLYMPIA implementation of this protocol appears in Figure 2 (client) and Figure 3 (server). The implementation proceeds as follows:

- **Round 1**: Each client broadcasts their public key (client lines 4-5).
- **Round 2**: Each client P_i generates n shares with threshold t of each element of their input vector x_i. P_i sends one share to each other client P_j (and keeps one for itself) (client lines 8-9).
- **Round 3**: Each client P_i receives 1 share of each other client P_j's input. P_i adds these shares together, to get one share of the sum of all clients' inputs. P_i sends the share of the sum to the server (client lines 12-14).
- **Round 4**: The server receives n shares of the total sum of the inputs, and reconstructs the total sum (server line 10).

C. Stevens et al. Protocol

Stevens et al. [32] design a protocol with similar structure to the secret sharing protocol, but leverage the learning with

Protocol	Setting	Client		Server		§
		Communication	Computation	Communication	Computation	
Secret sharing	—	$O(ln)$	$O(ln)$	$O(ln)$	$O(lnlog(n))$	IV-B
Stevens et al. [32]	Few clients	$O(l+k+n)$	$O(lk+nlogn)$	$O(ln+k)$	$O(lk+ln+nlogn)$	IV-C
Bonawitz et al. [5]	Few clients	$O(n+l)$	$O(n^2+nl)$	$O(n^2+nl)$	$O(ln^2)$	IV-D
Bell et al. [3]	Many clients	$O(logn+l)$	$O(log^2n+llogn)$	$O(nlogn+nl)$	$O(nlog^2n+nllogn)$	IV-E
Sharing sharing [31]	Many clients	$O(l+logn)$	$O(l+log^2n)$	$O(ln)$	$O(ln)$	IV-F
ACORN [2]	Many clients	$O(l+logn)$	$O(l+logn)$	$O(n(l+logn))$	$O(n(l+logn))$	IV-G

TABLE II: Communication and computation complexities of case study protocols implemented in OLYMPIA, for n parties aggregating vectors of size l and LWE security parameter k

```
1  class StevensClient(SecretSharingClient):
2    def round(self, round_number, message):
3      if round_number == 2:
4        S = self.GF.Random(self.s_len)
5        e = gen_noise()
6        A = GF.Random((self.dim, self.s_len), seed=seed)
7        masked_value = self.secret_input + A.dot(S) + e
8        self.secret_input = S  # secret share S vector
9        return self.masked_value,
10              ShamirClientAgent.round(self, round_number,
11                                      message)
12     else:
13       return ShamirClientAgent.round(self, round_number,
14                                      message)
```

Fig. 4: OLYMPIA implementation of the Stevens et al. [32] protocol (client).

errors (LWE) assumption [23] to reduce the dimensionality of the vector being secret shared to (effectively) a constant that depends on the security parameters.

The LWE assumption says that for a public matrix $A \in \mathbb{F}_q^{l \times s}$, secret vector $S \in \mathbb{F}_q^s$, and secret *error* vector $E \in \chi^l$, it is computationally hard to distinguish the pair (A, B) from a pair of uniformly random numbers, where $B = A \cdot S + E$, even when $s \ll l$.

Protocol description. The insight behind the Stevens et al. protocol is to use the vector $B \in \mathbb{F}_q^l$ as a *random mask* to send the secret input vector to the server, but aggregate the shorter S vector using secret sharing instead. This reduction in dimensionality leads to a corresponding reduction in communication cost.

Protocol implementation. The structure of the protocol matches that of the secret sharing protocol (Section IV-B) exactly. Because of the similarities, we can leverage the secret sharing protocol in our implementation, by subclassing the secret sharing protocol and modifying only round 2 (for the client) and round 4 (for the server). The client implementation appears in Figure 4. In round 2, the clients generate a random S_i (line 4) and secret share it (instead of their input vector x_i; line 7), and also send their masked vector $x_i + B_i$ to the server (line 8). In round 4, the server subtracts the aggregated masks after reconstructing $\sum_i S_i$. No other changes are necessary.

D. Bonawitz et al. Protocol

Bonawitz et al. [5] design a protocol with per-client communication cost of $O(n+l)$. The key idea behind this approach is to leverage *pairwise masking*: if client A adds a random *mask* to their input vector, and client B subtracts the *same* mask from their input vector, then summing up the two masked vectors eliminates the mask and yields the sum of the original vectors.

In the Bonawitz et al. protocol, each client adds such a mask to their vector for *each other client*. Adding up all of the masked vectors yields the sum of the original input vectors, and the server is prevented (by the masks) from learning any of the individual input vectors. To achieve reduced communication cost, clients agree on their pairwise masks by computing them from each other's public keys, by using the result of the *agree* function (Section IV-A) as the random seed for a pseudorandom number generator (PRNG) and using the PRNG to generate the mask vector.

We implemented the semi-honest and malicious variants of the Bonawitz et al. protocol in OLYMPIA. Our implementation uses the same Shamir secret sharing primitives and public-key encryption as the secret sharing baseline (Section IV-B). Per-client communication cost for the protocol includes the masked vector itself ($O(l)$), plus the client's public key ($O(1)$) and the Shamir shares of b_u and $s_{u,v}$ ($O(n)$).

E. Bell et al. Protocol

The communication complexity of the Bonawitz et al. protocol is nearly optimal for small sets of clients and large vectors, but its per-client communication cost is much higher than an insecure solution when n is large. The Bell et al. protocol [3] modifies the Bonawitz et al. protocol by relaxing the requirement that each client add a mask for each other client. Instead, in the Bell et al. protocol, each client adds a mask for a *subset* of the other clients.

The Bell et al. protocol generates a graph connecting clients to a set of k *neighbors* with whom they will exchanged pairwise masks. By setting $k = \log n$, the protocol reduces the number of masks in each masked vector from $O(n)$ to $O(\log n)$, and thus also reduces the per-client communication cost. The original Bonawitz et al. protocol can be recovered from the Bell et al. protocol by using a fully-connected graph.

We implemented the Bell et al. protocol in OLYMPIA by re-using large portions of our implementation of the Bonawitz et al. protocol. Our implementation also generates each client's set of neighbors, and uses the set of neighbors for each client to determine the construction of the masks. The per-client communication cost of our implementation is equivalent to that of our Bonawitz et al. implementation, but it requires only

$O(k)$ Shamir shares for the $s_{u,v}$ values, achieving the desired $O(\log n + l)$ complexity.

F. Secret Sharing Sharing Protocol

The Sharing Sharing protocol [31] builds on the Stevens et al. protocol by using a *two-level* secret sharing scheme. First, each client creates two additive shares of their input (the "high-level shares"); then, the clients split into two sets of groups of size $\log(n)$ and aggregate their high-level shares by running an aggregation protocol within the small groups. The clients reveal their group-level sums to the server, which cannot reconstruct any single input because of the higher level of additive sharing. This idea is combined with the LWE-based masking approach of the Stevens et al. protocol to reduce dimensionality, so that the combined protocol scales well with both the number of clients and the size of the input vectors.

We re-used large portions of our implementation of the Stevens et al. protocol to implement the sharing sharing protocol. The main differences are in the high-level sharing and group assignment for clients. The original paper specifies that clients themselves perform reconstruction of the group-level results; we offload this computation to the server instead, under the assumption that the clients will have limited computation power. We reduce the number of server-side reconstructions by adding shares together *before* reconstruction whenever possible.

G. ACORN Protocol

The ACORN protocol [2] combines the sparser communication graph of the Bell et al. protocol with a key-homomorphic approach for encoding both personal and pairwise masks based on LWE. As in the Stevens et al. protocol, this approach improves computation cost. Unlike Stevens et al., the ACORN protocol generates (homomorphic) encryption keys from public-key agreement, which reduces the dimensionality of the secret-shared data in the protocol. The ACORN protocol is part of a larger contribution that also includes input validation via zero-knowledge proofs.

Our implementation of ACORN leverages the structure of the Bell et al. protocol's implementation, and also re-uses some of the LWE-related implementation from the Stevens et al. protocol. We do not implement the input validation approach proposed in the paper. Rather than implement the paper's proposed packing scheme for plaintexts, we adjust the size of the finite field so that packing for plaintexts is not needed (since it lacks the input validation step, our implementation is not tied to the large finite field required for Bulletproofs).

V. Evaluation

In this section, we evaluate the concrete performance of the different secure aggregation protocols in terms of computation time, communication costs, scalability, and the effects of various parameters, such as the number of clients, input vector size, latency, etc. We designed our experiments specifically to answer the following questions:

- **RQ1** How well do the case study protocols scale with the size of the input vectors?
- **RQ2** How well do the case study protocols scale with the number of clients?
- **RQ3** What is the effect of network latency on protocol performance?
- **RQ4** What is the effect of network bandwidth limits on protocol performance?
- **RQ5** What is the overhead of malicious security?
- **RQ6** Does Olympia's simulator yield accurate results when compared to actual "ground truth" execution?

Setting	Protocols	Clients	Dimensions
Large Vectors	All	64	$1e^{[1,2,3,4,5]}$
Few Clients	All	$2^{[3,4,5,6,7]}$	100
Many Clients	Bell, Sharing sharing, ACORN	[100, 1000, 3000, 5000, 10000]	100

TABLE III: Experiment Settings.

A. Experiment Setup

Our experiments use Olympia to evaluate the concrete performance of the protocols described in Section IV. Our open-source release contains the code for both the Olympia framework and the implementations of these protocols. We ran each experiment on a single machine with 32GB of memory, utilizing a high-performance cluster to execute multiple experiments simultaneously.

We split our experiments into two settings, following two common use cases. In the "Large Vectors" setting, we fixed the number of clients (to 64) and varied the size of the vectors being aggregated. In the "Few Clients" setting, we fixed the vector size to 100 and varied the number of clients from 8 to 128. In the "Many Clients" setting, we fixed the vector size to 100 and varied the number of clients from 100 to 10,000, in order to test the support of the Bell, ACORN, and Sharing Sharing protocols for a large number of clients. The input vectors for each protocol do not affect performance, so we used a constant input vector to allow for verifying the correctness of the output. We used a finite field of size $2^{31} - 1$. We report the average results and standard error over five runs.

In all of our experiments, we used the most favorable settings for each protocol that would ensure security in the semi-honest setting. Following Bell et al. [3], we set both the fraction of malicious clients and the fraction of dropouts to 5% of the total. For the Stevens et al. protocol, following [32], we set the size of the secret vector S to 710. For the Bell et al. and ACORN protocols, following the most favorable settings of [3], we set $k = 50$. For the sharing sharing protocol, we optimize group according to expected dropouts per group.

The Sharing Sharing protocol is not designed for situations with a few clients, and finding the optimal group size for best performance in these smaller settings is challenging. Hence, in our study, we have left out the results of this protocol for scenarios involving a small number of clients, focusing on its stronger performance in larger settings.

B. Accuracy of the simulation

Evaluating the accuracy of OLYMPIA's simulator is difficult by definition—OLYMPIA is designed to simulate evaluation scenarios that are typically not feasible on a large scale by other means. To address this and validate the reliability of OLYMPIA, we performed a physical deployment with a smaller number of clients, enabling us to obtain ground truth results. The primary use we envision for OLYMPIA is comparing different protocols or protocol configurations, either to demonstrate an advance in efficiency or in preparation for deployment of a protocol; for such comparisons, consistency and comparability of results is more important than absolute accuracy of running time. In addition, most of the key components for evaluation (e.g. computation time and size of transmitted messages) are measured directly by the simulator, so these results will be exactly comparable to protocol deployments on equivalent hardware.

In validating OLYMPIA, we constructed a real network backend using the same client and server classes as those implemented in the simulator. These experiments are conducted within an advanced high-performance computing environment, which includes a SLURM job scheduling system with each node allocated 32 GB of memory. The experiments involve launching a server process and multiple client processes (varying in number and dimensions) across different nodes, thus enabling us to replicate and test our network protocols in a realistic environment. We run our experiments on the Acorn, Baseline, Bell, Bonawitz, Shamir Sharing, and Stevens protocols with 5, 10, and 20 clients, and dimensions ranging from 10 to 100,000. This approach enables a direct comparison of total running time, providing valuable insights into the performance of the protocols under study in both simulated and real-world network environments.

Figure 5 reveals that when comparing simulated results to ground truth results, there is a strong correlation in the performance trends of the various protocols. The simulator exhibits a slight tendency to underestimate total running times, especially in scenarios with fewer clients and smaller dimensions. However, it consistently captures the essential performance and scalability trends of these protocols across different settings. Importantly, this correlation becomes more pronounced as the number of clients increases, with the simulation results for protocols aligning more closely with the ground truth. This trend confirms the simulator's ability to model complex network behaviors accurately, showcasing OLYMPIA's effectiveness in simulating diverse protocol behaviors.

C. Results: Semi-Honest Security

This section describes our comparison between semi-honest variants of the case study protocols.

Total time. Figure 6 summarizes the total running time of the case study protocols in the three experimental settings. In the Large Vector setting (Figure 6(a)), all protocols scale well with the dimensionality of the aggregated vectors; the Bell et al. and Bonawitz et al. protocols yield the lowest total running times. The others require matrix operations (by both the client and the server) that contribute to increasing computation time as vector size increases. In the Few Clients setting (Figure 6(b)), the Stevens et al. is slightly better than the Bell et al. and Bonawitz et al. protocols, due to the smaller vectors. In the Many Clients setting (Figure 6(c)), all three protocols scale well with the number of clients; in contrast to the other settings, the Sharing Sharing protocol yields the best concrete performance when the number of clients exceeds 3000.

Computation time. Figure 7 summarizes the server and client computation time for each protocol in each of the three settings. Comparing these results to Figure 6, it is clear that computation time is the most important factor in overall running time. In the Large Vectors setting (Figure 7(a)), server computation time is largest for the Stevens et al., ACORN, and Sharing Sharing protocols (due to the combination of matrix operations and reconstructions required on the server); client computation time is largest for the LWE-based protocols due to matrix operations. The same trend continues in the other settings—in the Few Clients setting (Figure 7(b)), the Bell et al. and Bonawitz et al. protocols perform best with server computation times for clients under 40, but for larger client counts, the Stevens et al. protocol becomes more efficient. In the Many Clients setting (Figure 7(c)), the Bell et al. and ACORN protocols have the lowest client computation time, but the Sharing Sharing protocol has a much lower server computation time, which results in the low total time seen earlier.

Network latency. Our previous experiments use OLYMPIA's model of network latency derived from actual internet speed test data. To answer RQ3, we varied the latency model to evaluate the impact of latency on total running time. Figure 8 presents the results. All of our case study protocols have fairly low round complexity, so even a huge increase in latency (from 0ms to 1000ms per message) does not cause a large increase in total running time. These results suggest that the case study protocols are likely to perform well even over high-latency WAN connections.

Network bandwidth limitations. Our previous experiments assume that bandwidth is unlimited, both for the server and for the clients. To evaluate the importance of bandwidth limitations for protocol performance, we evaluated each protocol with a 1mbps limit on client bandwidth and with a 1mbps limit on server bandwidth. The results appear in Figure 9. Limiting client bandwidth has little impact on total running time, but limiting server bandwidth has a large effect on all protocols.

This effect is directly linked to the amount of traffic the server receives. The total bytes received by the server appear in Figure 10. The amount of traffic received by the server increases more quickly with vector size for the Sharing Sharing protocol than the others (Figure 10(a), (b), and (c)). Complete results for communication cost, including the traffic sent and received by clients (which mirrors the traffic received by the server) appear in Appendix B.

D. Results: Malicious Security

To evaluate the cost of malicious security, we re-ran our experiments with the malicious-secure variants of all our case study protocols. The total running time results appear in Figure 11. The graphs are similar to the corresponding results in the semi-honest case, supporting the claim that the malicious-secure variants of these protocols incur only modest overhead. More precise results appear in Table IV, for the case

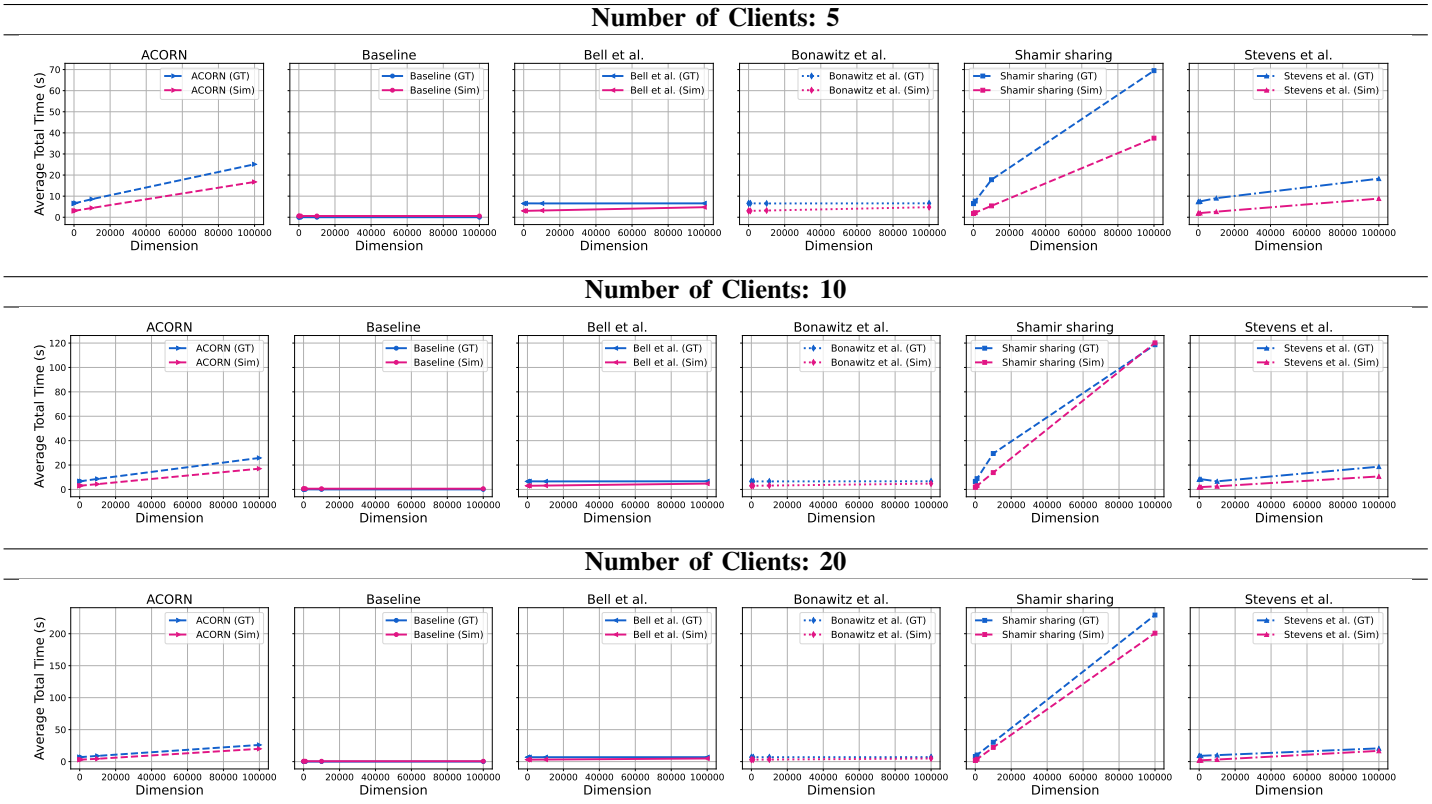

Fig. 5: Total running time comparison between simulated and real experiments.

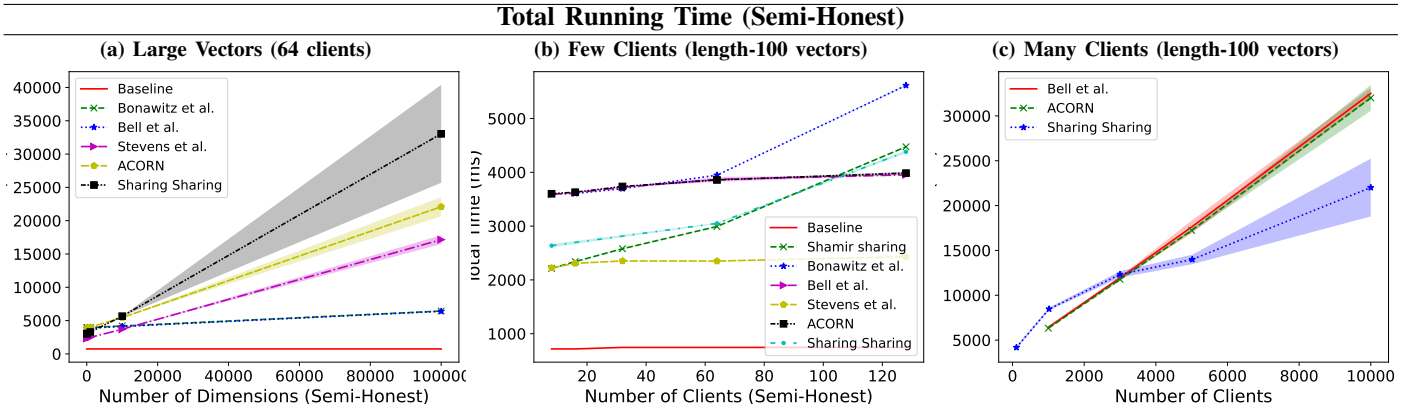

Fig. 6: Total running time comparison between semi-honest protocols. Shaded area indicates standard error.

Protocol	Semi-honest	Malicious	Overhead
Bonawitz et al.	2938ms	3795ms	29%
Bell et al.	3879ms	3938ms	1.5%
Stevens et al.	4104ms	5930ms	44%
ACORN	5649	5796ms	2.6%
Sharing Sharing	10269ms	23572ms	129%

TABLE IV: Precise results comparing semi-honest and malicious-secure variants of protocols, for 64 clients and length-10,000 vectors.

of 64 clients and length-10,000 vectors. Both sets of results suggest that the overhead of malicious security is reasonable for all protocols, and especially good for the Bell et al. and ACORN protocols. The Sharing Sharing protocol performs poorly in this setting. Additional results for malicious-secure protocols appear in Appendix C.

E. Discussion

Protocol comparison. Our results show that all the protocols have the expected asymptotic performance, but they differ significantly in terms of concrete performance. For a small number of clients (**RQ1**), the Bonawitz et al. and Bell et al. protocols provide the lowest total time. For a very large number

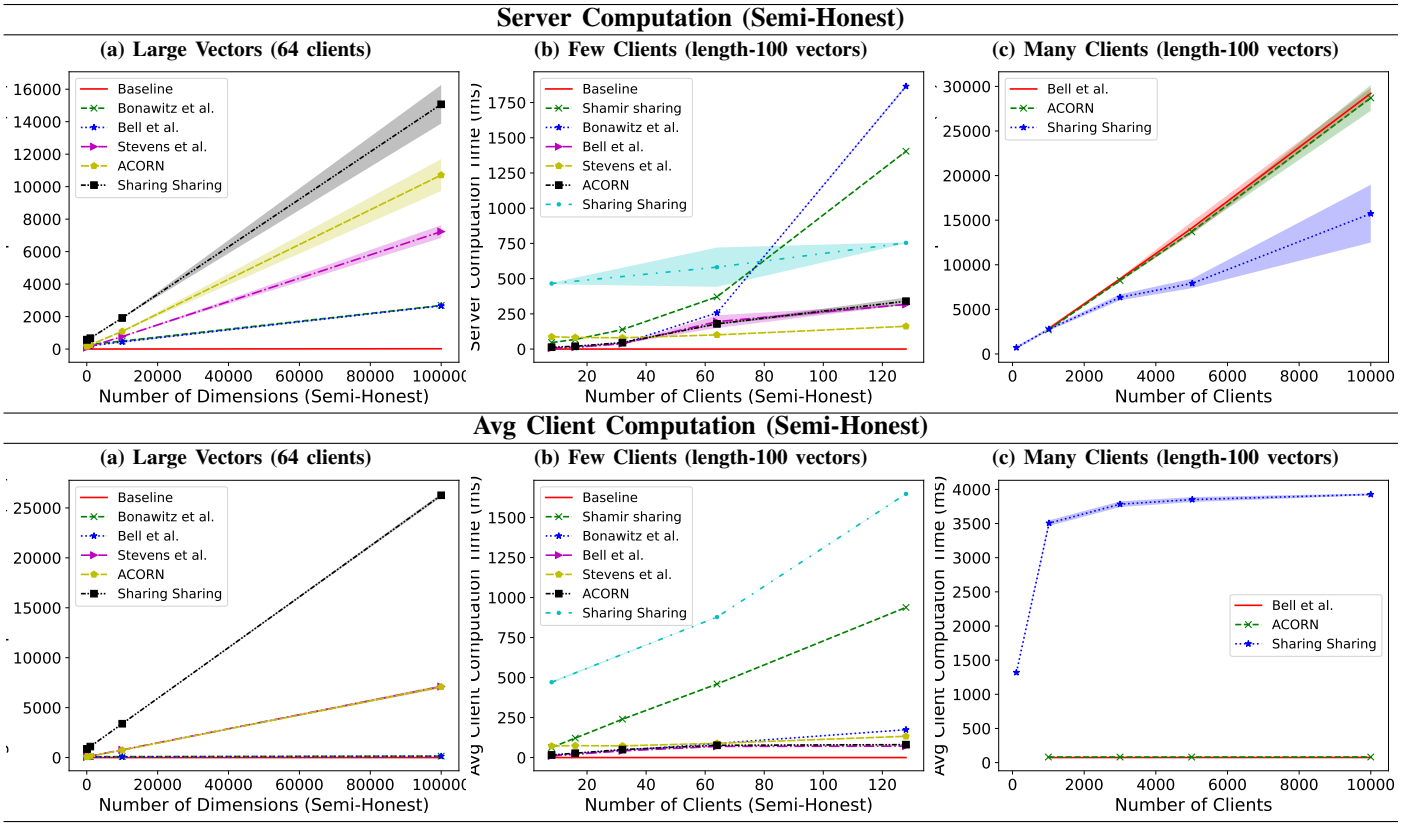

Fig. 7: Computation time comparison between semi-honest protocols. Shaded area indicates standard error.

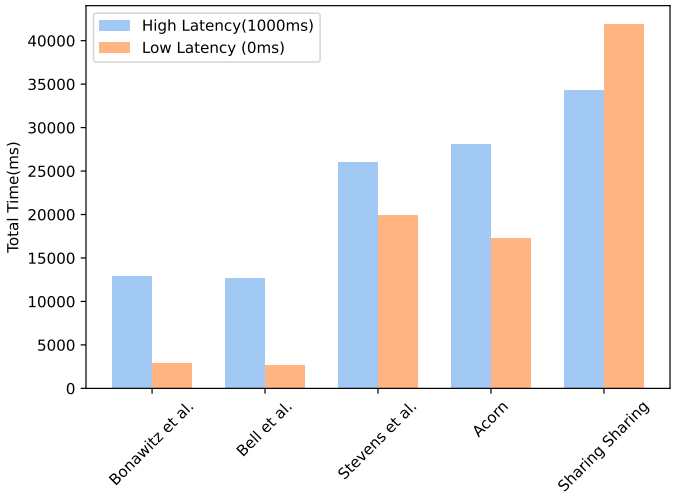

Fig. 8: Effect of latency on total running time (semi-honest security). Latency impacts all protocols in roughly the same way, since all protocols have similar round complexity.

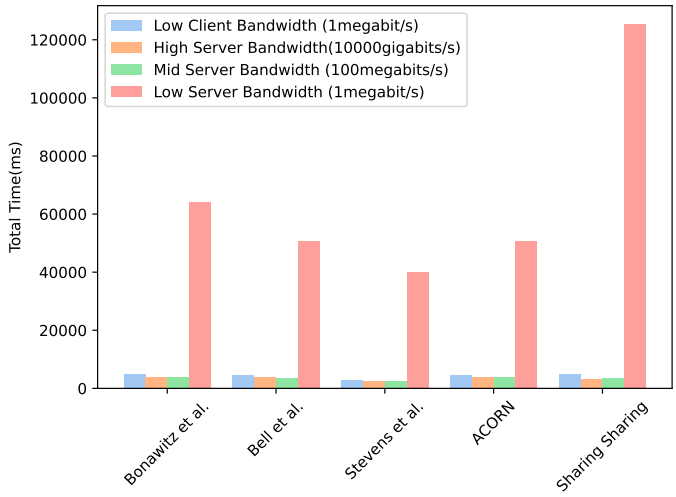

Fig. 9: Effect of bandwidth limitations on total running time (semi-honest security), with 64 clients and length-100 vectors. Limits on client bandwidth have a small impact, but limits on server bandwidth have a large impact.

of clients (**RQ2**), the Sharing Sharing protocol provides the best performance; the Bell et al. protocol may outperform Sharing Sharing for larger vectors.

Computation vs. communication. In all settings, the results suggest that computation time is the largest contributing factor to total running time of the protocol (**RQ3**). All of our case study protocols involve a small constant number of communication rounds and attempt to minimize total communication cost. Secure aggregation protocols with similar properties are likely to follow the same pattern.

Computation and constant factors. However, our results

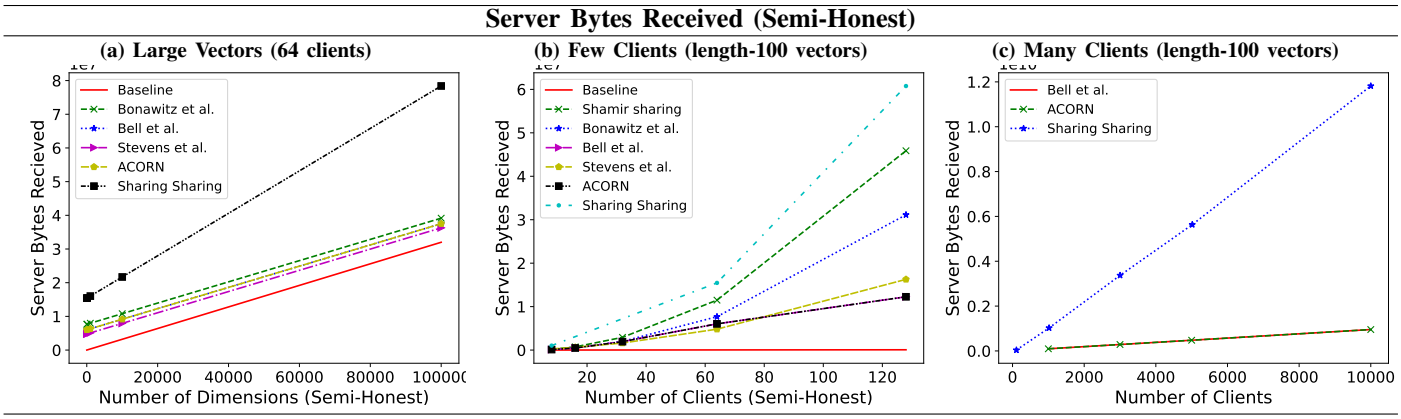

Fig. 10: Server bytes received comparison between semi-honest protocols.

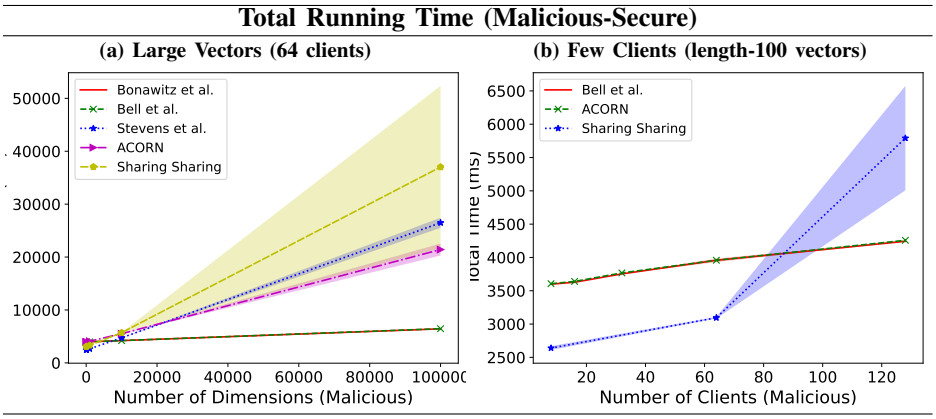

Fig. 11: Total running time comparison between malicious-secure protocols. Shaded area indicates standard error.

suggest that different protocols can have very different computation costs, and can distribute those costs between the clients and the server differently. For example, LWE-based approaches like Stevens et al., ACORN, and Sharing Sharing have higher computational costs for clients than the Bonawitz et al. and Bell et al. protocols, but tend to have lower computational costs for the server when vectors are small. The Sharing Sharing protocol in particular has much higher computation cost for the client, but much lower cost for the server in the Many Clients setting. For large vectors, however, the computation cost of the matrix operations required in these approaches means that the Bell et al. and Bonawitz et al. approaches often provide better performance. These results point to the importance of end-to-end evaluation of protocols in determining overall performance—such effects are typically impossible to completely understand only by analyzing the asymptotic behaviors of protocols, and depend on the deployment scenario.

Importance of latency (RQ3). The results of our network latency experiment suggest that typical WAN latencies are unlikely to have a major impact on the total running time of our case study protocols (**RQ3**). This result suggests that protocol design should focus on concrete performance improvements in computation time, rather than on further reducing round complexity.

Importance of bandwidth (RQ4). The results of our network bandwidth limitation experiment suggest that limits on client bandwidth are not likely to have significant impact on the running time of the case study protocols, but that limits on server bandwidth can have a huge effect (**RQ4**). Protocols differ slightly in the impact of server bandwidth limitations, but all are affected. These results suggest that deployments of all of these protocols must include adequate server bandwidth to ensure good performance.

Overhead of malicious security (RQ5). Our comparison between semi-honest and malicious variants of the case study protocols shows that the overhead of malicious security is below 50% for all protocols, and below 5% for the Bell et al. and ACORN protocols. These results suggest that malicious security can be achieved with good performance.

Feasibility of end-to-end evaluation (RQ6). Our evaluation results demonstrate the feasibility of end-to-end empirical evaluation of protocols with OLYMPIA, even at scales well beyond the feasibility of traditional evaluation. Our experiments involved up to 10,000 clients and vectors with millions of elements; these experiments ran on a single machine with 128GB of memory in just a few hours. Our validation experiment shows OLYMPIA's simulator accurately models the performance and scalability properties of protocols, and that its accuracy improves with the number of clients involved with the protocol—probably because the impact of execution over-

head (e.g. constructing and managing sockets and serializing messages) is smaller as the number of clients increases.

VI. Related Work

Secure aggregation. As described earlier, *secure aggregation* protocols are lightweight multiparty computation protocols specifically designed for summing up large vectors, and are primarily designed to facilitate federated learning applications.

The first practical protocol for the large-vector setting was due to Bonawitz et al. [5], while the first protocol for the many-client setting was due to Bell et al. [3]. Since then, several other protocols have been developed that make additional improvements. Among these, only Turbo-Aggregate [30] attempts to improve performance in the many-client setting; it reconstructs masks among subsets of users (rather than pairwise), but has a weaker threat model than the Bell et al. [3] protocol.

In the large-vector setting, several new protocols have been recently proposed, including MicroFedML [17] (reduces round complexity for sparse gradients), LightSecAgg [35] and the protocol of Stevens et al. [32] (improve concrete performance for dropouts), FastSecAgg [19] (improves performance using Fast Fourier Transform). All of these protocols can be simulated in Olympia, and are important targets for future empirical evaluations.

Input validation. Recent work has made progress towards ensuring input validity in secure aggregation. EIFFeL [26] requires each client to produce a zero-knowledge proof that their input is within a reasonable range; ACORN [2] improves on the performance of EIFFeL by leveraging more efficient aggregation of these proofs. Protocols that integrate input validation have even more complicated concrete performance properties and are an exciting future target for evaluation with Olympia.

Secure machine learning. Many approaches have been developed for machine learning *without* secure aggregation [33], [15], [28], [11], [18]. These approaches often ask clients to secret-share model updates between two servers, and run an MPC protocol between the two servers. These approaches have a much weaker threat model than secure-aggregation-based approaches, since they require non-collusion between the two servers. In addition, they do not present the same challenges for empirical evaluation as secure aggregation protocols, since only the two servers need to be simulated; while these approaches could be simulated using Olympia, a simulation-based approach is not required for empirical evaluation of these approaches.

General-purpose MPC. General-purpose MPC protocols evaluate arithmetic or boolean circuits, and therefore can (in principle) implement any function [36], [4], [21], [1], [10]. Such protocols are generally designed for two parties (2PC), three parties (3PC), or a small number of parties (MPC)—none are designed for the many-client setting. General-purpose MPC protocols can therefore be empirically evaluated without the use of a simulation framework like Olympia, though for recent work on larger-scale protocols (e.g. up to 128 parties [34] or even tens or hundreds of thousands [16]), Olympia may make implementation and evaluation simpler.

Other applications of secure aggregation. Several systems have been developed for differentially private *analytics* (i.e. database queries) that leverage ideas from secure aggregation, including Honeycrisp [24], Orchard [25], and Cryptϵ [27]. These systems are designed to scale to millions of participants, using specialized protocols and a slightly weaker threat model. Because they are designed for the many-client setting, these systems are also challenging to evaluate empirically; Olympia may also be useful for evaluation of such systems.

Simulation of distributed protocols. For protocols designed for a small number of clients (including most general-purpose MPC protocols), experimental evaluation is feasible using actual hardware. As described earlier, this kind of evaluation is essentially impossible in the many-client setting. The Bonawitz et al. protocol [5] included an experimental evaluation of a Java implementation, which was possible due to the relatively small number of clients. Due to its scale, the later Bell et al. protocol [3] included concrete results only for the number of neighbors each client must communicate with, and did not include an experimental evaluation of computation or communication cost. Similarly, Honeycrisp [24] and Orchard [25] include concrete experimental results for parts of their protocol, but do not perform end-to-end experiments due to the protocol's intended scale.

The ABIDES framework [6] was originally designed to simulate agents in a financial market at large scale. It has been previously applied in an *ad-hoc* manner to experimentally evaluate few-client secure protocols [7], [17], [8]. Olympia builds on ABIDES to provide a framework for designing, building, evaluating, and refining secure protocols, with a particular focus on the many-client setting.

VII. Conclusion

We have presented Olympia, a framework for designing, building, evaluating, and refining secure aggregation protocols. Olympia enables experimental evaluation with thousands of clients—a setting in which evaluation on actual hardware is not possible. The Olympia framework provides a simulator that accurately measures the end-to-end running time of protocols, and includes a domain-specific language (DSL) embedded in Python for defining synchronous secure aggregation protocols. We have used Olympia to conduct an empirical comparison between several existing protocols implemented in our case studies, yielding new insights about the concrete performance of these protocols. We release the Olympia framework and our case study implementations as open source, and hope that Olympia will be useful as a standardized implementation and evaluation tool for new protocols.

Acknowledgments

We thank Timothy Stevens for his help implementing the Stevens et al. protocol [32]. This material is based upon work supported by the National Science Foundation under Grant No. 2238442 and by DARPA under Contract No. HR001120C0087. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or DARPA. Computations were performed on the Vermont Advanced Computing Center supported in part by NSF award No. OAC-1827314.

REFERENCES

[1] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. Association for Computing Machinery.

[2] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. Acorn: Input validation for secure aggregation. *Cryptology ePrint Archive*, 2022.

[3] James Henry Bell, Kallista A Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1253–1269, 2020.

[4] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. Association for Computing Machinery.

[5] Kallista Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[6] David Byrd, Maria Hybinette, and Tucker Hybinette Balch. Abides: Towards high-fidelity market simulation for ai research. *arXiv preprint arXiv:1904.12066*, 2019.

[7] David Byrd, Vaikkunth Mugunthan, Antigoni Polychroniadou, and Tucker Hybinette Balch. Collusion resistant federated learning with oblivious distributed differential privacy. *arXiv preprint arXiv:2202.09897*, 2022.

[8] David Byrd and Antigoni Polychroniadou. Differentially private secure multi-party computation for federated learning in financial applications. In *Proceedings of the First ACM International Conference on AI in Finance*, pages 1–9, 2020.

[9] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*, pages 259–282, 2017.

[10] Ivan Damgard, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority – or: Breaking the spdz limits. Cryptology ePrint Archive, Report 2012/642, 2012. https://ia.cr/2012/642.

[11] Alex Davidson, Peter Snyder, E. B. Quirk, Joseph Genereux, and Benjamin Livshits. Star: Distributed secret sharing for private threshold aggregation reporting, 2021.

[12] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), 1976.

[13] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[14] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3), 2017.

[15] Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Helen Möllering, Thien Duc Nguyen, Phillip Rieger, Ahmad-Reza Sadeghi, Thomas Schneider, Hossein Yalame, et al. Safelearn: secure aggregation for private federated learning. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 56–62. IEEE, 2021.

[16] S Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale mpc. In *Advances in Cryptology–EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part II*, pages 694–723. Springer, 2021.

[17] Yue Guo, Antigoni Polychroniadou, Elaine Shi, David Byrd, and Tucker Balch. Microfedml: Privacy preserving federated learning for small weights. *Cryptology ePrint Archive*, 2022.

[18] Bargav Jayaraman, Lingxiao Wang, Katherine Knipmeyer, Quanquan Gu, and David Evans. Revisiting membership inference under realistic assumptions. *Proceedings on Privacy Enhancing Technologies*, 2021(2), 2021.

[19] Swanand Kadhe, Nived Rajaraman, O Ozan Koyluoglu, and Kannan Ramchandran. Fastsecagg: Scalable secure aggregation for privacy-preserving federated learning. *arXiv preprint arXiv:2009.11248*, 2020.

[20] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.

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

[22] Ookla. Speedtest® by Ookla® Global Fixed and Mobile Network Performance Maps. Based on analysis by Ookla of Speedtest Intelligence® data for 2020. https://www.kaggle.com/datasets/dhruvildave/ookla-internet-speed-dataset.

[23] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[24] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. Honeycrisp: large-scale differentially private aggregation without a trusted core. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 196–210, 2019.

[25] Edo Roth, Hengchu Zhang, Andreas Haeberlen, and Benjamin C Pierce. Orchard: Differentially private analytics at scale. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1065–1081, 2020.

[26] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. Eiffel: Ensuring integrity for federated learning. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2535–2549, 2022.

[27] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. Cryptϵ: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 603–619, 2020.

[28] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing, 2020.

[29] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[30] Jinhyun So, Başak Güler, and A Salman Avestimehr. Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning. *IEEE Journal on Selected Areas in Information Theory*, 2(1):479–489, 2021.

[31] Timothy Stevens, Joseph Near, and Christian Skalka. Secret sharing sharing for highly scalable secure aggregation. *arXiv preprint arXiv:2201.00864*, 2022.

[32] Timothy Stevens, Christian Skalka, Christelle Vincent, John Ring, Samuel Clark, and Joseph Near. Efficient differentially private secure aggregation for federated learning via hardness of learning with errors. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1379–1395, 2022.

[33] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 1–11, 2019.

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

[35] Chien-Sheng Yang, Jinhyun So, Chaoyang He, Songze Li, Qian Yu, and Salman Avestimehr. Lightsecagg: Rethinking secure aggregation in federated learning. *arXiv preprint arXiv:2109.14236*, 2021.

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

APPENDIX A
ADDITIONAL EVALUATION RESULTS: SHARING SHARING PROTOCOL

Figure 12 contains the full results for all semi-honest protocols, including the Sharing Sharing protocol. This protocol is excluded from the Large Vectors and Few Clients settings in the main body of the paper because it is not competitive, and including it makes the other protocols more difficult to compare.

APPENDIX B
ADDITIONAL EVALUATION RESULTS: SEMI-HONEST SECURITY

Figure 13 contains additional results for the semi-honest variants of the case study protocols: the average bytes sent and received by clients, and the bytes sent by the server. These results mirror those presented in Section V—the server sends roughly as much traffic as it receives, and each client sends and receives roughly $1/n$ of the traffic sent and received by the server—because all communication between clients is routed through the server.

APPENDIX C
ADDITIONAL EVALUATION RESULTS: MALICIOUS SECURITY

Figures 14 and 15 contain additional results for the malicious-secure variants of the case study protocols. These results mirror those for the semi-honest variants presented in Section V, and are included here for completeness.

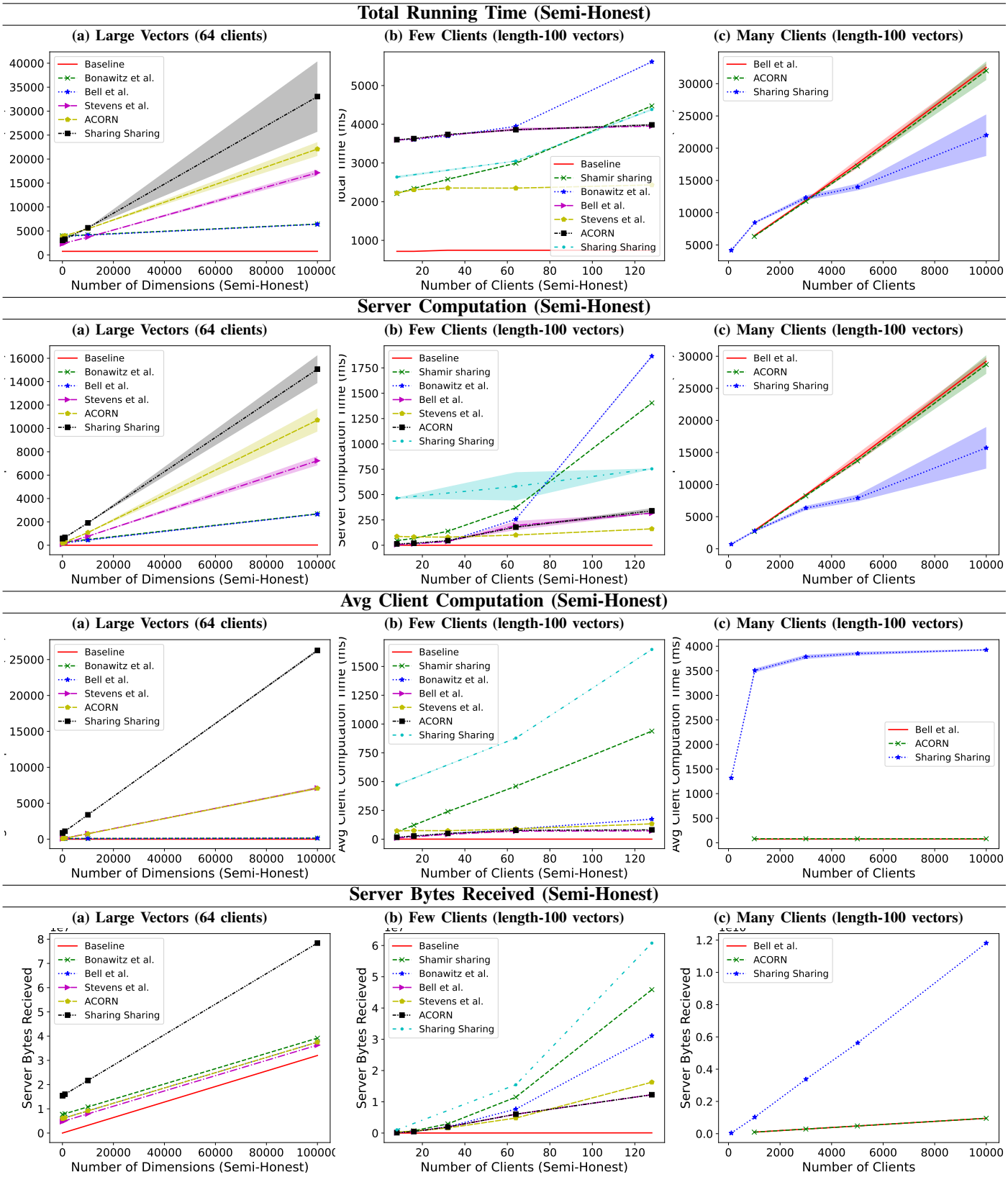

Fig. 12: Comparison between semi-honest protocols, including Sharing Sharing. Shaded area indicates standard error.

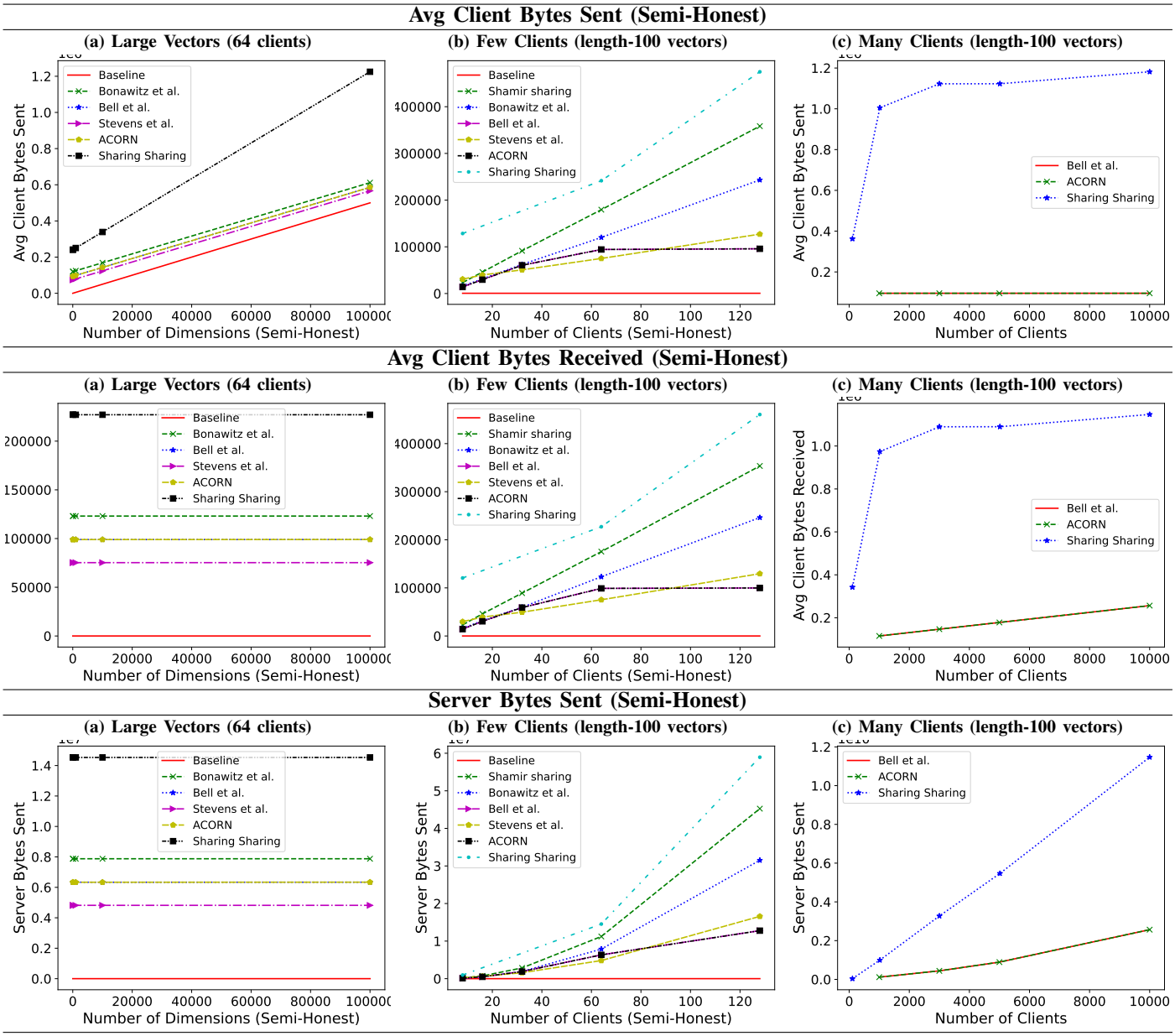

Fig. 13: Additional results for semi-honest protocols.

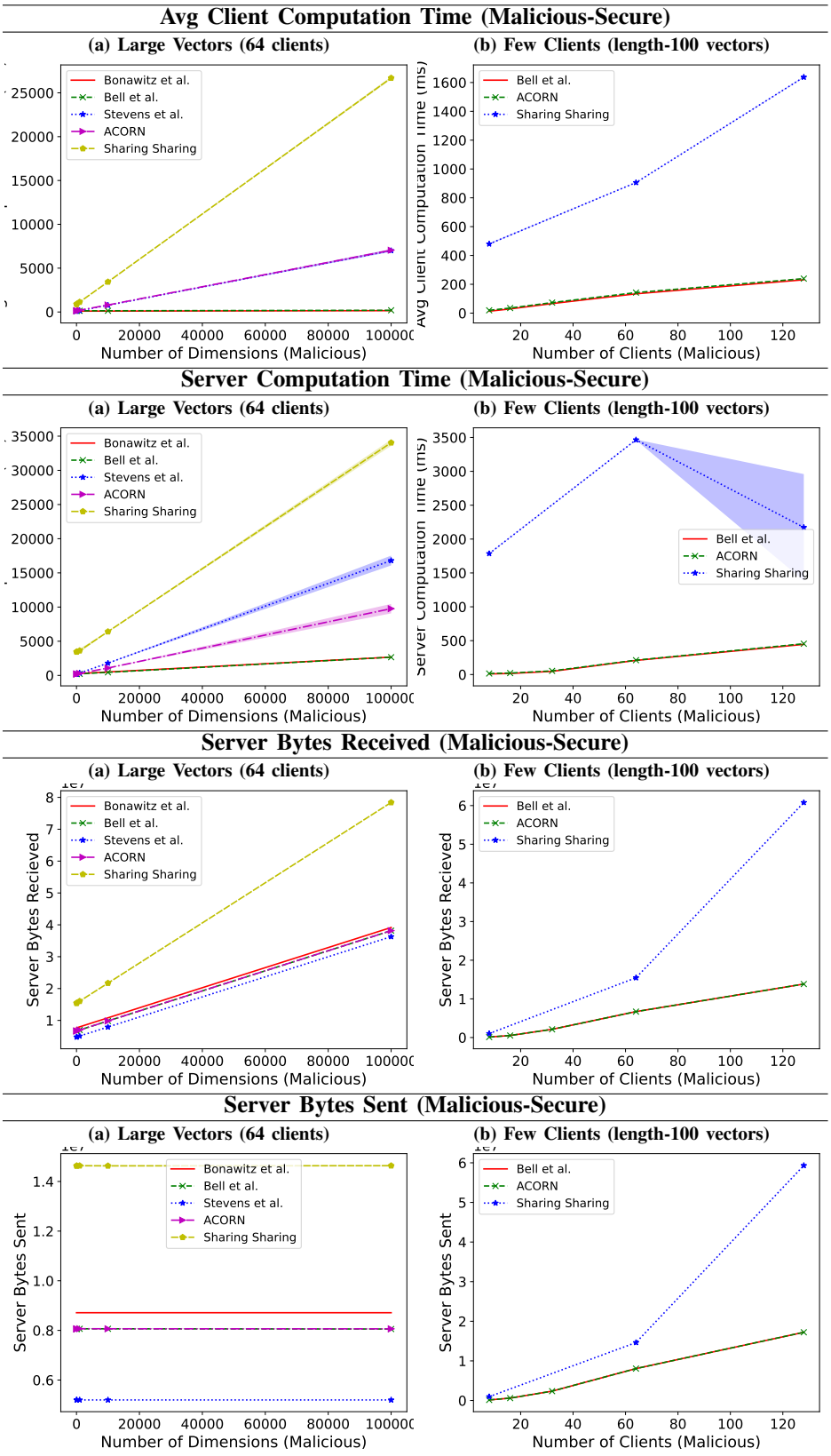

Fig. 14: Additional results for malicious-secure protocols.

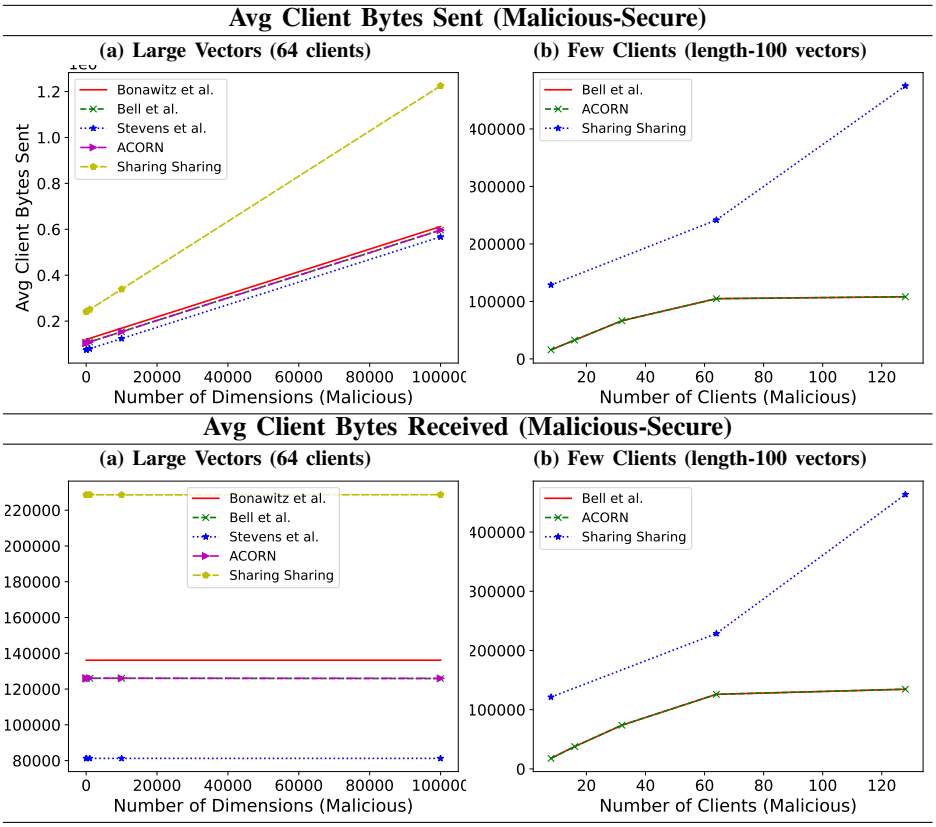

Fig. 15: Additional results for malicious-secure protocols.