Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares

Surya Addanki [1], Kevin Garbe [2], Eli Jaffe [3], Rafail Ostrovsky [4], and Antigoni Polychroniadou [5]

[1,2,3,4]UCLA
{*surya7, kgarbe, jaffe, rafail*}*@cs.ucla.edu*
[5]J.P. Morgan AI Research
antigoni.poly@jpmorgan.com

Abstract

This paper introduces Prio+, a privacy-preserving system for the collection of aggregate statistics, with the same model and goals in mind as the original and highly influential Prio paper by Henry Corrigan-Gibbs and Dan Boneh (USENIX 2017). As in the original Prio, each client holds a private data value (e.g. number of visits to a particular website) and a small set of servers privately compute statistical functions over the set of client values (e.g. the average number of visits). To achieve security against faulty or malicious clients, Prio+ clients use Boolean secret-sharing instead of zero-knowledge proofs to convince servers that their data is of the correct form and Prio+ servers execute a share conversion protocols as needed in order to properly compute over client data. This allows us to ensure that clients' data is properly formatted essentially for free, and the work shifts to novel share-conversion protocols between servers, where some care is needed to make it efficient. While our overall approach is a fairly simple observation in retrospect, it turns out that Prio+ strategy reduces the client's computational burden by up to two orders of magnitude (or more depending on the statistic) while keeping servers costs comparable to Prio. Prio+ permits computation of exactly the same wide range of complex statistics as the original Prio protocol, including high-dimensional linear regression over private values held by clients.

We report detailed benchmarks of our Prio+ implementation and compare these to both the original Go implementation of Prio and the Mozilla implementation of Prio. Our Prio+ software is open-source and released with the same license as Prio.

1 Introduction

In recent decades, modern society has exploded with a wave of internet-enabled devices. Smartwatches, cell-phones, cars, and ATMs are constantly collecting data on their surroundings to improve performance. For many cloud services controlling such devices, collecting and computing statistics over such a large pool of data has become a hugely profitable endeavor. Navigation apps detect congestion with user location data [24], fitness trackers collect average data for user comparison [22]. Aggregate statistics are one of the principle currencies in the modern data-driven economy.

Although these services only wish to compute aggregate statistics, not collect individual data, their methods often involve storing users' personal data in the clear on their servers and then computing statistics directly on that data. Such a centralized cache of sensitive user data presents clear security risks. As described in [8], a motivated attacker may simply steal and disclose this sensitive information [26, 35], cloud services could misuse or sell this information for profit [34], and intelligence agencies may acquire the data for targeting or mass surveillance purposes [20].

The specific problem to be solved is as follows: each of n clients P_i holds a private value $x_i \in \{0, 1\}$, and they wish to learn the sum $\sum_i x_i$. As described in the original Prio paper [8], some previous systems have also attempted to solve this problem of privately computing aggregate statistics. One such attempt involves using a randomized response system to provide differential privacy [17, 19]. That is, some user data is replaced with random data according to some fixed probability $p < 0.5$. By aggregating this "noisy" data, data collectors can get a somewhat accurate estimate of overall statistics. This technique scales well and provides robustness (each malicious client can at most affect the sum by ± 1), but the privacy guarantees are relatively weak. There is an inherent tradeoff between the privacy guarantee to the client and the accuracy of the overall statistic. Another option is to have clients submit encryptions of their data to servers. Then, servers can sum up the ciphertexts and only decrypt the final sum [13, 15, 25, 27, 1, 28]. This achieves stronger privacy guarantees but sacrifices robustness: a malicious client can affect the final sum arbitrarily because servers cannot tell the difference between an encryption of 0/1 and an encryption of some large integer. Such attacks are often incentivized: if used for a voting scheme, this would allow any client to submit as many votes as they like. These attacks can be mitigated using zero-knowledge proofs [16], but such approaches heavily impact scalability. Servers require expensive public-key operations to verify these proofs, and clients are burdened with the computationally difficult task of generating the proofs.

Prio is a brilliant and highly influential private aggregation system which successfully resolves this discrepancy between privacy, robustness, and scalability. Prio works within the client-server model in which the n clients rely on a small number of computationally powerful servers in order to compute aggregate statistics. Prio provides relatively strong privacy guarantees: it guarantees privacy so long as at least one of the computation servers is honest. It also provides robustness: any malicious client can affect the protocol no more than misreporting their private data value as another valid value. That is, if the client is supposed to submit a value in the range $[0, 64]$, Prio servers can syntactically reject submissions of any value outside that range, but clients can of course choose to submit a different value in that range besides their true private value. To achieve this, Prio utilizes a new technique called SNIPs (secret-shared non-interactive proofs), which allows servers to collaboratively check a shared proof of correctness at a low communicational cost. In

particular, the bandwidth used by servers during verification remains constant as the size of user inputs increases. The Prio protocol has been widely adopted, and has even been re-implemented by Mozilla for use in privately collecting web usage statistics. It is being run as a service for other web-based organizations by the Internet Services Research Group (ISRG), a non-profit focused on reducing barriers to secure communication over the internet. Google has also begun using it to perform analytics in their exposure notifications express (ENX) system for measuring health data.

Prio achieves highly desirable security guarantees, and their solution achieves significant efficiency gains over other comparably secure data-collection systems. However, the client-side computation and client-to-server communication of their solution, though better than other comparable systems, increases linearly with the size of user data. In the client-server model, client computation and communication costs are most often the bottlenecks for overall efficiency, since clients are on low-power devices and high-latency connections whereas servers are usually collocated high-power machines. For example, clients usually run on either web browsers or cell phones when using the Mozilla Firefox browser. It is somewhat inefficient then to require clients on such devices to use SNIPs to verify simple properties like the size of a user's input, as it places an unnecessarily large computational burden on these weak devices.

In this paper, we present Prio+, a new and improved version of Prio which is optimized to reduce overall burden on the client, not the servers. Prio+ utilizes a Boolean secret-sharing scheme so that clients can prove their data falls within the correct range at essentially zero computational cost. Then, servers simply execute a Boolean-to-arithmetic share conversion protocol (if necessary) to continue computing statistics as in the original Prio protocol. In some cases we utilize an existing share conversion protocol, but we also develop a novel share conversion protocol using extended doubly authenticated bits (edaBits) [18] for use in situations where existing protocols do not suffice. For some of our protocols on particular statistics, Prio+ still uses SNIPs, as they are particularly well-suited for verifying certain multiplicative relationships. We do not, however, use them to verify everything about client inputs as is done in Prio. We implemented Prio+ in C++ and our code is publicly available at `https://github.com/KuraTheDog/Prio-plus`.

Our strategy significantly reduces both client computation and client-to-server communication, the two most expensive computational resources in our efficiency model. Even for the few statistics where SNIPs are still necessary (variance and linear regression), the size of the SNIPs and the work necessary for clients to create them decreases dramatically. The result is a system which computes the same set of complex statistics as Prio with identical privacy and robustness guarantees but with reduced client computation and client-to-server communication. For example, when collecting the distribution of tens of thousands of client responses to a simple true/false question, Prio+ clients can encode their data over 350x faster than Prio clients, and the client's message size is nearly 5x smaller. In many cases, Prio+ also improves server efficiency: For the example given above, once servers have received all client inputs, Prio+ servers are able to process client submissions 85x faster with essentially the same server-to-server communication. As the size of inputs increases, Prio+ sees an increase in server communication whereas Prio's server communication stays constant. But, for practically sized inputs, Prio+ servers still communicate only a few hundred bytes per client submission.

	Client Mults	Proof Size	Server Mults	Serv Sym Key	Serv Comm.
Prio	$M \log M$	M	$(M \log M)n$	None	n
Linear PCPs [**5**]	$M \log M$	$\log M$	$(M \log M)n$	None	n
Π_{sum}	None	None	L	L	$n + L^2$
$\Pi_{\mathsf{and/or/max/min}}$	None	None	None	None	n
$\Pi_{\mathsf{var/linReg}}$	$M' \log M'$	M'	$(M' \log M' + L)n$	L	$n + L^2$
Π_{frq}	None	None	nL	L	$(n + L) \log n$

Table 1: Table of asymptotic comparisons between Prio and the protocols comprising Prio+. M is the number of multiplication gates required to check if an input is properly encoded for the relevant statistic. L is the bit-length of each client's input, $M' := M - L$ is the number of multiplication gates not used for checking bit-length, and n is the total number of clients. 'Serv Sym Key' refers to the symmetric-key operations necessary for servers to perform Boolean-to-arithmetic share conversion via the methods of [14]. 'Serv Comm.' is the communication between servers in bits. $\Theta(\cdot)$ notation suppressed to improve readability.

Both Prio and Prio+ support computation of more complex statistics in addition to just summation. While Prio applies a general SNIP-based solution across the board, Prio+ applies a specialized approach for each complex statistic and uses SNIPs for a lighter relations as needed. For example, the Prio+ summation protocol, the protocol which is most heavily used by Mozilla, does not require SNIPs at all. This specialized approach means the efficiency of Prio+ varies depending on the statistic being computed. For some such statistics, Prio+ not only improves client performance and server computation, but additionally achieves extremely low and constant server communication. As an example, when computing the maximum of client data in the range $[0, 128]$, Prio+ servers communicate a constant 16 bytes per client, compared to a constant 740 bytes for Prio servers. This is in addition to a nearly 750x faster client encode time, 5x smaller client message size, and a 43x improvement in server computation time. Even for the few statistics where Prio+ still utilizes SNIPs, we see significant improvements in client encode time, client message size, and server compute time at little-to-no server bandwidth cost.

Contributions: In what follows we summarize our contributions:

- Provide a detailed edaBit-based semi-honest Boolean-to-arithmetic share conversion protocol whose output shares lie in a field $\mathbf{Z}_p$, which was not explicit in [18],
- demonstrate how to use Boolean-to-arithmetic share conversion in conjunction with Boolean secret-sharing and smaller-scope SNIPs for particular relations in order to provide robustness and privacy in a large-scale data collection system, and
- demonstrate that client usage of Boolean representation avoids expensive Zero-Knowledge proofs which leads to dramatic speed-ups of the system overall, and
- exhibit the effectiveness of our protocols with a full-scale and publicly available implementation allowing private and robust computation of a wide range of complex statistics.

We now give an informal theorem of our results for a single protocol (Π_{sum}). We establish similar results for the remaining protocols according to the asymptotics given in Table 1.

Theorem 1. *(Informal) Suppose n players $P_1, \ldots, P_n$ each hold a private L-bit integer x_i, and they wish to rely on two servers S_L and S_R to compute the sum $f(x_1, \ldots, x_n) = \sum_i x_i$. There exists a*

protocol Π_{sum}*, returning the sum* $f(\cdot)$ *to each client and returning no output to either server, which is both private and robust against a coalition of up to* n *malicious clients and one semi-honest server and requires zero client multiplications, no zero-knowledge proofs,* $O(L)$ *multiplications per server,* $O(L)$ *symmetric key operations per server, and* $O(n)$ *bits of communication between servers.*

With Prio+, we hope to provide the same benefits as Prio to systems and organizations whose clients cannot withstand the burden of generating and sending expensive zero-knowledge proofs.

2 Technical Overview

In this section we briefly overview the remaining sections of the paper.

Arithmetic vs. Boolean Secret-Sharing: Secret-sharing (specifically, threshold secret-sharing) is a cryptographic tool which allows a user to "share" a private value x into a vector of values $[x] = ([x]_1, \ldots, [x]_n)$ in such a way that any strict subset of these values reveals nothing about x, but all values together can be used to reconstruct x completely. Prio is primarily built around arithmetic secret-sharing: $[x]_i$ are random values in a ring $\mathbf{Z}_M$ (often this is a field and $M = p$) subject to the constraint $\sum_i [x]_i = x \pmod M$. Clients share their private inputs and send one share to each server. Servers then sum them up locally and then return those local aggregations to clients. Clients them combine the local aggregations to learn the total sum. Servers force clients to submit private values within a particular range by requiring clients to also submit specialized zero-knowledge proofs attesting to that fact.

Another possible secret-sharing scheme is called Boolean secret-sharing. Here, the client holds a private L-bit integer x and secret-shares it as $([x]_1, \ldots, [x]_n)$, where each $[x]_i$ is a random L-bit integer subject to the constraint $\bigoplus_i [x]_i = x$, where the direct sum is done component-wise with each bit of the binary representation. This scheme obeys many of the same properties as arithmetic secret-sharing, particularly that each share appears random except when combined with all other shares. The crucial difference is the following: if each share $[x]_i$ is an L-bit integer, it is guaranteed that the private value x is also an L-bit integer. This allows servers to verify the bit-length of a client's submitted private value via simple local checks on the shares themselves, without needing expensive zero-knowledge proofs. Note: to distinguish Boolean shares from arithmetic shares, arithmetic shares will be denoted $[x]^A$, and Boolean shares will be denoted $[x]^B$.

Boolean-to-Arithmetic Share Conversion: Replacing arithmetic secret-sharing with Boolean secret-sharing presents one important issue: how do servers sum up the clients' private values if they only have Boolean shares of those values? The original idea of summing shares locally and then returning those aggregated values to clients only works due to associativity of addition: $([x]_1^A + [y]_1^A) + ([x]_2^A + [y]_2^A) = ([x]_1^A + [x]_2^A) + ([y]_1^A + [y]_2^A) = x + y$. Since Boolean shares need to be reconstructed via componentwise-XOR, not standard addition, no such associativity trick applies. Instead, Prio+ servers will utilize a Boolean-to-arithmetic share conversion protocol, which takes as input Boolean shares $([x]_1^B, \ldots, [x]_k^B)$ and outputs a random arithmetic sharing $([x]_1^A, \ldots, [x]_k^A)$ of the same underlying value x. Such protocols have been studied extensively in the past [14, 18, 9], and the most efficient method based on oblivious transfer (OT) is due to [14]. The resulting arithmetic shares are in the ring $\mathbf{Z}_{2^l}$ for some integer l. This is sufficient for computing the sum since there is no reason for the arithmetic shares to necessarily lie in a field. However, when we compute

some specific complex statistics using Prio+, we will rely on some properties of a field when using arithmetic shares, which will require a different Boolean-to-arithmetic share conversion protocol. This particular problem of outputting arithmetic shares in a field $\mathbf{Z}_p$ has not been studied explicitly to our knowledge, and thus we devise our own protocol based on extended double-authenticated bits (edaBits) [18]. Other approaches to this problem seem feasible: for example, recent results in function secret-sharing (FSS) allowing secure comparison of secret-shared values may suffice. These alternative methods, however, seem to impose more overhead than the edaBit-based approach in order to achieve the desired functionality.

Semi-Honest Boolean-to-arithmetic Share Conversion into $\mathbf{Z}_p$ via edaBits: edaBits are a newly studied cryptographic primitive based on double-authenticated bits (daBits) [18]. A daBit is a secret-shared vector $([b]^A, [b]^B)$ where $b \in \{0, 1\}$ is a random bit unknown to all parties. It is well-known how to use daBits for Boolean-to-arithmetic share conversion. Converting an L-bit integer requires L daBits in total, which is relatively expensive and motivated the construction of extended double-authenticated bits (edaBits). An L-bit edaBit is a secret-shared vector $([r]^A, [r_0]^B, \ldots, [r_L]^B)$ where r is a random L-bit integer and $r = \sum_i 2^i r_i$. It was discussed in [18] that one can use an L-bit edaBit for Boolean-to-arithmetic share conversion similarly to how one uses L individual daBits, but that construction was not made explicit. Particularly relevant to our applications, if $[r]^A$ is shared in the field $\mathbf{Z}_p$, then the shares outputted by the share conversion protocol will also lie in $\mathbf{Z}_p$. We reiterate the process of generating edaBits and also detail the steps of using edaBits for Boolean-to-arithmetic share conversion in the semi-honest model in Section 9.

Multiplication Triples: There are a few points within the protocol which require servers to multiply secret-shared values. It is well-known that this can be achieved by consuming multiplication triples, which are secret-shared vectors of the form $([a], [b], [ab])$ where a, b are random values. In the original Prio protocol, such multiplication triples are used during SNIP verification. Normally, multiplication triple generation is expensive, although this fact is offset by the fact that they are independent of user data and can thus be generated offline during low-traffic times. Prio's multiplication triples are particularly cheap to generate because the client is allowed to generate them. Amazingly, even if a malicious client supplies an incorrect triple, this will not trick the servers into approving an invalid SNIP. This is due to a beautiful argument regarding the polynomial testing procedure, which is discussed in detail in [8]. In Prio+, generating edaBits for the Boolean-to-arithmetic share conversion protocol also requires multiplication of secret-shared values. The vast majority of these values are single bits. Multiplying single bits does not require multiplication triples: we can simply use oblivious transfer (OT). We utilize silent OT extension [6] so as to minimize server-to-server communication. However, to ensure that the edaBits are the correct bit-length, a single double-authenticated bit (daBit) is necessary for each edaBit generated, and generating a daBit requires multiplying large secret-shared values. This is the one step in which our protocol uses multiplication triples, and it requires one multiplication triple in the field $\mathbf{Z}_p$. To generate these few multiplication triples we utilize a semi-honest version of the method from [11] which is based on homomorphic encryption. Servers randomly generate their shares of a and b and then use the multiplicative property of the encryption scheme to compute shares of the product ab. This method has the added benefit of allowing servers to pack many shares into the same ciphertext, which enables generation of many multiplication triples at once with a single round of communication. This is not necessarily the most efficient method: recent advances

in pseudorandom-correlation generators (PCGs) suggest the possibility of producing these triples more cheaply [7]. Since we require so few of them, however, we chose to use standard homomorphic encryption-based methods to enable easier implementation of the resulting protocol.

Complex Statistics: Prio+ supports computation of the exact same statistics as Prio. In particular, in addition to SUM, clients can compute Boolean AND / OR (where each client holds a single bit), MAX / MIN and frequency count FRQ (where each client holds a value in some small range $[0, K]$), integer variance VAR and standard deviation STDDEV (where each client holds an L-bit integer), and linear regression linReg (where each client holds a degree d feature vector of L-bit integers).

Many of these statistics (AND, OR, MAX, MIN) are even simpler than SUM, requiring no share conversion, no zero-knowledge proofs, and virtually no communication between servers. FRQ, similar to SUM, requires share conversion to allow summation on the server side, but does not require any zero-knowledge proofs. Instead, servers use some simple logical mechanisms to detect improperly encoded inputs, which we will discuss in the next section.

The only statistics which do require some zero-knowledge proofs are VAR, STDDEV, and linReg. In these cases, clients encode their private values in such a way that the most efficient method for verifying that encoding is to use secret-shared non-interactive proofs (SNIPs) similarly to Prio. The key difference is that the SNIPs in this case are only being used to verify a very specific part of the encoding, whereas in Prio they are used to verify every property of the encoded value. For example, when computing VAR, clients should submit values of the form $\vec{x} = (x, x^2)$ where x is an L-bit integer. Prio would use one large SNIP to verify both of those facts at once. By contrast, Prio+ first uses Boolean secret-sharing to ensure that $\vec{x}_1$ is an L-bit integer and $\vec{x}_2$ is a $2L$-bit integer. Then, we use SNIPs to verify the relation $\vec{x}_1^2 = \vec{x}_2$. Because the construction of SNIPs is centered around verifying the output of multiplication gates, they are particularly well-suited to verifying multiplicative relations of this sort which require relatively few multiplications. Because SNIPs operate on arithmetic shares of the input, we first apply edaBit-based Boolean-to-arithmetic share conversion on the clients' submitted Boolean shares. The reason OT-based share conversion from [14] does not suffice is that SNIP verification is based on a batched polynomial identity test [8] which is only valid if the underlying polynomial is defined over a field.

Frequency Count: To compute FRQ, P_i is required to submit shares of an impulse vector δ_{x_i} with value 1 at index x_i and value 0 at all other locations. Servers then sum up these impulse vectors component-wise to output a histogram $\vec{h}$ of the distribution of client values. Verifying that no client submitted a non-impulse vector is somewhat non-trivial: it requires more than simply verifying the length of individual shares, but checking the relation via SNIPs requires many multiplication gates since it is not an inherently multiplicative relation. However, servers can accomplish this task at relatively low cost using some simple logic. First, servers use the Boolean shares of client inputs to check the parity of the number of 1's in each client's submitted vector. This requires passing a single bit per client and reveals nothing about any honest client's input, since an honest input will always be an impulse and thus have an odd number of 1's. This step ensures that no client submitted a zero vector, since that would have an even number of 1's. Next, servers check whether the total number of 1's is equal to the total number of players. If so, this (in combination with the

fact that no player submitted a zero vector) implies that all players submitted a single impulse. If not, they can locate the misbehaving player by repeating this check a logarithmic number of times on smaller subsets of the players, honing in on the misbehaving player via binary search. To check whether the total number of 1's is equal to the total number of players, servers simply apply Boolean-to-arithmetic share conversion on each component of each shared vector, locally sum all components of all shared vectors, recombine the total sum and check this against the total number of players. After this check identifies all misbehaving players and those inputs are discarded, servers sum the arithmetic shares of all well-behaving clients and return these summed vectors to the clients who sum them together to get the histogram $\vec{h}$.

Practical Comparison: We compared Prio+ to both the original Go implementation of Prio as well as another implementation by Mozilla which only computes SUM, no complex statistics. We did not compare to the updated construction given in [5] because it has not yet been implemented and is focused on the case of extremely long inputs. In the case of practically-sized inputs, large constant terms overshadow the asymptotically smaller client message sizes.

First, we compared all three implementations in evaluating the $\mathsf{SUM}(x_1, \ldots, x_n)$ where $n = 10,000$, $n = 50,000$, and $n = 100,000$. We recorded the encode time (milliseconds per client), client message size (bytes per client per server), server compute time (milliseconds per server per client), and server communication (bytes per server per client) and averaged the results from each value of n. We ran four separate experiments in which x_i was 1-bit, 8-bits, 16-bits, and 32-bits respectively. Prio+ clients were able to encode their data up to 540x faster than in the Go implementation of Prio, and up to 3000x faster than in the Mozilla implementation. Client messages in Prio+ were over 3x smaller than in the Mozilla implementation, and up to 23x smaller than in the Go implementation. Although reducing server computation time was not the primary goal of this project, Prio+ servers processed client submissions up to 116x faster than the Go implementation and up to 615x faster than the Mozilla implementation. The expected drawback of these savings was server communication, since conserving server bandwidth was the primary motivation for Prio's SNIPs. Prio+ does see increased server bandwidth in some cases, particularly as the size of user inputs increases. However, the practical bandwidth usage for 1-bit integers is essentially the same as in the Mozilla implementation and 18x less than the Go implementation. By comparison, for 32-bit inputs, Prio+ server communication is still 3x less than the Go implementation, and just 7x more than the Mozilla implementation. This tells us that for practically-sized user inputs, Prio+ achieves monumental improvements in client computation, client communication, and server computation with minimal impact to server communication.

We also ran similar experiments between Prio+ and the Go implementation of Prio for MAX and linReg. Since MAX requires no SNIPs and no share conversion, we saw improvements across the board: when client values lied in the range $[0, 128]$, Prio+ client encode time was 750x less, client message size was 5x smaller, server compute time was 43x less, and server communication was 46x less. Even though linReg still requires some SNIPs (with reduced scope), we saw up to 30x lower client encode time, up to 4x smaller client message size, up to 3x less server compute time, and server communication varying between 5x less (for degree 2 inputs) and 1.5x more (for degree 8 inputs). Prio+ is clearly more efficient across the board when computing MAX and low-degree linReg. For higher-degree linReg, we see significant gains in encode time, client message size, and

server compute time for a slight increase in server communication.

3 Preliminaries

In this section we will describe our computational model. This includes a description of our ideal functionality, the client/server setup, our efficiency model, the set of adversaries we defend against, and the assumptions we rely upon to build our protocol.

"Two-Party" Setting: In this work we construct protocols for secure computation of a wide range of aggregate statistics in the client-server model. That is, a set of n clients with private data wish to compute statistics on that data with the help of two honest-but-curious servers. The basis of our system is a secure two-party protocol between these servers. That is to say that each client with an input, secret-shares his/her input between the two computation servers (which are assumed to not collude). Then, the two computation servers run the secure two-party computation protocol on the input shares which does not reveal to either server any information about client inputs. Finally, they send the output shares back to the clients who then reconstruct the output. This technique of using secure two-party computation in the client-server setting was first described in the ABY framework of [14] and has been used in many applications since, including [8].

Just as in the ABY framework of [14], this also allows for reactive computations in which the two computation servers maintain some secure state information between multiple executions. This could be used in the case where client data is being collected over time and the data is sent to the servers one point at a time.

Formally, our deployment consists of n clients $\{P_1, \ldots, P_n\}$ and 2 servers S_L and S_R. Each client P_i holds some private input x_i. Each client can communicate with each server and servers can communicate with clients and each other via private channels, but clients do not communicate with each other. Side note: although Prio+ is described as a two-server protocol, it can be easily generalized to a $2s$ server protocol for any positive integer s by simply running Prio+ s times in parallel with each pair of servers collecting data from their designated area.

Efficiency Model: The complexity measures we consider are computation and communication of each type of party (or pair of parties). In particular:

- Computation per client, τ_c
- Computation per server, τ_s
- Client-to-server communication, ρ_c
- Server-to-server communication, ρ_s

Instead of optimizing for low communication or computation overall, we assume clients have low computational power and servers have high computational power. Similarly, we assume a low-bandwidth connection from clients to servers, and a high-bandwidth connection between servers. Thus we seek to minimize τ_c and ρ_c as our highest priority, and τ_s and ρ_s as an afterthought. In general, we assume network latency is the greatest bottleneck in computation, so for both clients

and servers we are concerned more with optimizing their communication than their computation.

Security Against Semi-Honest Servers, Malicious Clients: Our deployment protects client *privacy* as long as at most one server is passively corrupted (regardless of malicious client misbehavior). Our system cannot tolerate malicious, misbehaving servers as this comes at a direct cost of functionality, as discussed in [8]. Our deployment always provides *robustness* (correctness) so long as neither server maliciously misbehaves. We summarize our security definitions here, please refer to Appendix A for details.

Privacy: Intuitively, our deployment provides f-privacy for an aggregation function (statistic) f if an adversary controlling any number of clients and all but one server learns nothing about the honest clients' inputs besides what is revealed by the output of f. More formally, any such adversary can simulate its view of the protocol run given the output of f. For some aggregation functions, we weaken our protocol to provide $\hat{f}$-privacy where $\hat{f}$ leaks slightly more information than the statistic itself (for example, servers may leak the number of clients who provided invalid inputs).

Robustness: A protocol is t-robust if a coalition of t malicious clients cannot affect the output of the protocol beyond misreporting their private data values. This is the strictest notion of correctness in the malicious security model, since a client's private input is known to nobody but themselves, meaning we cannot prevent them from misreporting it. If this data value is meant to come from a specific domain, however, malicious clients should *not* be able to submit data from outside of that domain. This is particularly relevant in our setting, where client data must be encoded correctly to permit efficient computation of the aggregate statistic. Clients should absolutely not be able to submit improperly encoded data, as this allows a single client to affect the output arbitrarily without detection. Our deployment is robust against malicious clients, but not against malicious servers. Though robustness against malicious servers may seem desirable, it comes at a direct cost to performance, as argued in [8]. Since the number of clients is much larger than the number of servers, it is much more reasonable to prevent and/or replace faulty servers than faulty clients.

Analogously to [8], we assume cryptographic primitives for the establishment of pairwise authenticated channels (CCA-secure public key encryption [10], digital signatures [32, 33], etc.). We make no synchrony assumptions about our network and do not rely on external systems to provide users anonymity.

Notation: We write $x \oplus y$ to denote the XOR operation (addition modulo 2), $x +^l y$ for addition within the ring $\mathbf{Z}_{2^l}$, and $x +^p y$ for addition in the field $\mathbf{Z}_p$. When $\vec{x}, \vec{y} \in \mathbf{Z}_{2^m}$ are vectors of bits, we will write $\vec{z} = \vec{x} \oplus \vec{y}$ to denote the bitwise-XOR operation. That is, $(\vec{z})_i = (\vec{x})_i \oplus (\vec{y})_i$ for each $0 \leq i < m$. We assume a maximum bit-length l on all integer data and thus treat all integer-valued data as elements of the ring $\mathbf{Z}_{2^l}$, except in the case where we perform share conversion on client data. In that case, the resulting shares will lie within the field $\mathbf{Z}_p$, as this is required for our polynomial identity testing procedure. This of course requires that $p > 2^l$, which will always be the case. We denote an arithmetically secret-shared variable x by $[x]^A$. A variable x shared in the Boolean secret-sharing scheme is denoted $[x]^B$. We will exclusively use two-party secret-sharing, and thus shares held by server S_L will be written $[x]^t_L$, $t \in \{A, B\}$. Shares held by server S_R will similarly be written $[x]^t_R$. For an integer x, we refer to the i'th least significant bit of the binary

representation of x as $(x)_i$. We will say that a function $f : \mathbf{N} \longrightarrow \mathbf{R}$ is negligible if for every positive polynomial poly there exists an integer N_{poly} such that for $x > N_{\mathsf{poly}}$, $|f(x)| < \frac{1}{\mathsf{poly}(x)}$.

4 Necessary Primitives

In this section we describe the necessary building blocks for our construction. We outline the details of the Arithmetic and Boolean secret-sharing schemes and we also give a short description of Beaver's private multiplication procedure for arithmetically secret-shared values, first detailed in [3], which plays a crucial role in our system.

First, we review the general concept of secret-sharing. We are motivated by a client holding secret value x who wishes to "share" this secret value between N different servers. A general N-party t-threshold secret-sharing scheme consists of two algorithms:

> $\mathsf{Share}(x)$ takes a secret x and returns shares $[x]_1, \ldots, [x]_N$
>
> $\mathsf{Rec}(\{[x]_i\})$ returns x if and only if $\{[x]_i\}$ consists of at least t different shares produced by $\mathsf{Share}(x)$.

Here we give a general definition in terms of t and N, but our protocols will all adopt the values $t = N = 2$.

A secret-sharing scheme must have two properties: privacy and correctness. Privacy in this setting means any set of less than t shares of x reveals nothing about x. Correctness means that the function Rec succeeds on every valid set of t or more shares. Each of these conditions must hold except with negligible probability

We now describe the Arithmetic and Boolean secret-sharing schemes, which are central to our construction. These schemes are described in [14], please refer there for further details.

Definition 4.1. Arithmetic Secret-Sharing: Given an integer $x \in \mathbf{Z}_M$, an arithmetic secret-sharing of x is a random pair $a, b \in \mathbf{Z}_M$ subject to the condition $a + b = x \pmod M$. We will first formally describe the sharing semantics. Then we will describe the process of adding shared values as well as multiplying shared values via the use of arithmetic multiplication triples, a.k.a Beaver triples [3].

Semantics: The two-party arithmetic secret-sharing scheme consists of the following pair of functions:

- $\mathsf{Share}_{+,M} : \mathbf{Z}_M \longrightarrow (\mathbf{Z}_M)^2$, $\mathsf{Share}_{+,M}(x) = ([x]^A_L, [x]^A_R)$, which are random elements of $\mathbf{Z}_M$ subject to the constraint $[x]^A_L + [x]^A_R = x$.
- $\mathsf{Rec}_{+,M} : (\mathbf{Z}_M)^2 \longrightarrow \mathbf{Z}_M$, $\mathsf{Rec}_{+,M}([x]^A_L, [x]^A_R) = [x]^A_L + [x]^A_R \pmod M$.

Operations: Every efficiently computable function can be represented as a circuit of multiplication and addition gates, which can each be performed over arithmetically secret-shared values as follows:

Addition/Scalar Multiplication: $[z]^A = [x]^A + [y]^A$. Each server $i \in \{L, R\}$ locally computes $[z]_i^A = [x]_i^A + [y]_i^A$. Similarly, to compute scalar multiplication $[w]^A = c \cdot [x]^A$ for $c \in \mathbf{Z}_M$, each server locally computes $[z]_i^A = c \cdot [x]_i^A$.

Multiplication: $[z]^A = [x]^A \cdot [y]^A$. Servers use a pre-computed Arithmetic multiplication triple (a.k.a Beaver triple) [3] of the form $([a]^A, [b]^A, [c]^A)$, where $c = ab$ and $a, b, c \in \mathbf{Z}_M$. This triple is consumed in order to perform the multiplication according to the following protocol Mult^A. In general, this protocol works for an arbitrary number of players, but we present it as a two-party protocol between S_L and S_R so as not to cause confusion.

Definition 4.2. The protocol Mult^A takes as input $([x]^A, [y]^A, [a]^A, [b]^A, [c]^A)$. That is, it begins with each server $i \in \{L, R\}$ holding an arithmetic share $[x]_i^A$ of some secret x and an arithmetic share $[y]_i^A$ of some secret y, and also holding arithmetic shares $([a]_i^A, [b]_i^A, [c]_i^A)$, where $c = ab \pmod M$. It computes arithmetic shares $[z]^A$ of $z = xy \pmod M$ as follows:

- Each S_i broadcasts $[x]_i^A - [a]_i^A$ so that every party can compute $x - a$ in the clear. They similarly broadcast $[y]_i^A - [b]_i^A$ so that each party learns $y - b$ in the clear.
- An arbitrary party (say S_L) outputs $z_1 = c_1 + (x - a)b_1 + (y - b)a_1 + (x - a)(y - b)$
- S_R outputs $z_i = c_i + (x - a)b_i + (y - b)a_i$.

It is proven in [3] that $\mathsf{Mult}^A(\cdot)$ correctly outputs $[z]^A$, an arithmetic secret-sharing of the product $z = xy \pmod M$.

Definition 4.3. Boolean Secret-Sharing Given an integer $x \in \mathbf{Z}_{2^l}$, a Boolean secret-sharing of x is a random pair $c, d \in \mathbf{Z}_{2^l}$ subject to the condition $c \oplus d = x$.

Semantics: The two-party l-bit Boolean secret-sharing scheme consists of the following pair of functions:

- $\mathsf{Share}_{\oplus,l} : \mathbf{Z}_{2^l} \longrightarrow (\mathbf{Z}_{2^l})^2$, $\mathsf{Share}_{\oplus,l}(x) = ([x]_L^B, [x]_R^B)$, which are random elements of $\mathbf{Z}_{2^l}$ subject to the constraint $[x]_L^A \oplus [x]_R^A = x$.
- $\mathsf{Rec}_{\oplus,l} : (\mathbf{Z}_{2^l})^2 \longrightarrow \mathbf{Z}_{2^l}$, $\mathsf{Rec}_{\oplus,l}([x]_L^A, [x]_R^A) = [x]_L^A \oplus [x]_R^A$.

Operations: Any efficiently computable function can be represented as a circuit of XOR and AND gates, which can be computed over Boolean secret-shares as follows:

XOR: $[z]^B = [x]^B \oplus [y]^B$. Each server $i \in \{L, R\}$ locally computes $[z]_i^B = [x]_i^B \oplus [y]_i^B$.

AND: $[z]^B = [x]^B \wedge [y]^B$. This is computed using two instances of OT, details can be found in Section 9 ($\Pi_{\mathsf{edaBitGen}}$ step 3(a)(iii)).

These two secret-sharing schemes have different strengths and weaknesses. When client data is shared under the Boolean scheme, servers can locally extend l'-bit shares of some secret value to l-bit shares ($l > l'$) of the same secret value without sacrificing privacy. They do this by simply appending $l - l'$ leading zeroes to their personal shares. This means that servers can require client inputs to be at most l' bits, verify that each share they received has length l', and then lift those shares into equivalent shares in a larger domain to compute on them without overflowing the modulus. In contrast, lifting arithmetic shares into a larger domain is non-trivial and quite expensive [14]. On the other hand, if client data is arithmetically secret-shared under a large modulus M, servers can efficiently compute the sum of shared values by locally summing their shares modulo M and sending these aggregated shares back to the client for reconstruction, as utilized in many previous works, including Prio [8]. We will soon show how our system leverages both of these advantages in order to allow clients to more efficiently and privately compute complex aggregate statistics. In particular, Prio+ clients will submit their data in the Boolean scheme (so that servers can verify the bit-length efficiently) and then servers will convert these shares back to the arithmetic scheme in order to sum the data together and compute the given statistic.

Boolean to arithmetic share conversion is a well-studied technique. The current most efficient protocol in the semi-honest two-party setting is due to [14] and is based on Oblivious Transfer (OT). In particular, to convert a pair of l-bit Boolean shares, they use l independent instances of OT where each OT transfers on average a string of length $(l + 1)/2$. The total communication cost is $l(\kappa + (l + 1)/2) = O(l^2)$ [14]. The resulting arithmetic shares lie in the ring $\mathbf{Z}_{2^l}$. The most efficient method for computing OTs in our model, and the method which we will utilize, is silent OT-extension [6] which minimizes communication between servers.

While this OT-based share conversion protocol suffices for some of our applications, computing more advanced statistics requires that the resulting arithmetic shares lie in a field. This is due to specific requirements of the zero-knowledge proofs we will use to prove inputs are well-formed. We describe these proofs in more detail in the following paragraph. To circumvent this issue, we develop and utilize a share conversion protocol whose output shares lie in the field $\mathbf{Z}_p$. Our protocol is based on a recently discovered primitive called extended doubly-authenticated bits, or edaBits [18]. Although this primitive has mostly been utilized in the setting of malicious security, we use them to construct an efficient share conversion protocol in the semi-honest setting. Our protocol is based on existing techniques but, as far as we know, has not yet been detailed explicitly. Our full protocol can be found in Section 8.

The final piece which some of our protocols utilize is the secret-shared non-interactive zero-knowledge proof (SNIP) which underpins the Prio protocol of [8], which we described briefly in the introduction. Although our position is that their system overuses SNIPs in situations where they are unnecessary, in the right circumstances SNIPs are an incredibly efficient method for verifying multiplicative relationships on secret-shared inputs in the client-server setting. We review here the method of using SNIPs to allow servers to efficiently verify that $\mathsf{Valid}(x) = 1$ for some client input x without learning any additional information. The following description comes directly from [8].

A secret-shared non-interactive proof (SNIP) protocol consists of an interaction between a client (the prover) and multiple servers (the verifiers). At the start of the protocol:

- Each server i holds a share $[x]_i \in \mathbf{F}^L$
- The client holds the secret input (vector) $x = \sum_i [x]_i \in \mathbf{F}^L$.
- All parties hold an arithmetic circuit representing $\mathsf{Valid} : \mathbf{F}^L \longrightarrow \mathbf{F}$.

The client's goal is to convince the servers that $\mathsf{Valid}(x) = 1$ without revealing any additional information about x. To do so, the client sends a proof string to each server. After receiving these proof strings, the servers gossip amongst themselves and then conclude either that $\mathsf{Valid}(x) = 1$ (accept x) or $\mathsf{Valid}(x) \neq 1$ (reject x).

A valid SNIP must satisfy correctness, soundness, and zero-knowledge.

> **Correctness.** If all parties are honest, the servers will accept x.
>
> **Soundness.** If all servers are honest, and if $\mathsf{Valid}(x) \neq 1$, then for all malicious clients, even ones running in super-polynomial time, the servers will reject x with overwhelming probability. In other words, no matter how the client cheats, the servers will almost always reject.
>
> **Zero-knowledge.** If the client and at least one server are honest, then the servers learn nothing about x, except that $\mathsf{Valid}(x) = 1$. More precisely, there exists a simulator (that does not take x as input) that accurately reproduces the view of any proper subset of malicious servers executing the SNIP protocol.

The construction in [8], based on a generalized version of the polynomial-based batched multiplication verification technique of Ben-Sasson et al. [4], satisfies each of these properties as proven in their Appendix D. We omit the details of their construction due to space limitations, but we emphasize that multiplication verification is central to their construction and thus it is well suited for verifying multiplicative relationships among inputs, as these can be verified using relatively few multiplication gates in the Valid circuit. This number of multiplication gates, which we denote μ, is the bottleneck efficiency parameter for their construction. Client-to-server communication is proportional to μ, and client computation is proportional to $\mu \log \mu$. For this reason, we restrict ourselves to using SNIPs only for verifying naturally multiplicative relations, as opposed to Prio which uses them universally for all types of input verification. For example: suppose a client is meant to submit shares of $\vec{x} = (x, x^2)$ where x is a b-bit integer. Prio's solution is to use one large SNIP to verify at once both the fact that $\vec{x}_1$ is a b-bit integer as well as the fact that $\vec{x}_1^2 = \vec{x}_2$. Prio+, on the other hand, would verify the bit-length of x using Boolean secret-sharing, and use a smaller SNIP to verify the naturally multiplicative relation $\vec{x}_1^2 = \vec{x}_2$.

5 The Simple Scheme

Our protocols are all based on the following simple scheme for computing the sum of clients' private bits. This is also the basis of Prio's solution and is described in [8]. We reiterate that scheme here for convenience.

Each client P_i, $i \in \{1, \ldots, n\}$, holds a private bit $x_i \in \{0, 1\}$. They wish to learn the sum $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$. Even this basic functionality has wide-ranging applications for data

collectors, since it allows one to survey clients on any yes/no question and learn the distribution of responses. Consider the following protocol for computing the sum.

1. **Upload**: Each P_i computes $\mathsf{Share}_{+,M}(x_i) = ([x_i]_L^A, [x_i]_R^A)$. The client then sends one additive share to each server over secure pairwise-authenticated channels. Note: although x_i is a single bit, we treat it here as an element of $\mathbf{Z}_M$ for $M > n$.
2. **Aggregate**: S_L and S_R hold accumulator values $A_L, A_R \in \mathbf{Z}_p$ respectively, initially set to zero. For each i, when S_L receives $[x_i]_L^A$ from P_i, computes $A_L \leftarrow A_L + [x_i]_L^A \pmod M$. S_R does the same with its accumulator A_R upon receiving $[x_i]_R^A$ from P_i.
3. **Publish**: Once data is collected, servers publish their accumulator values A_L, A_R to every client.
4. **Client Computation**: Clients compute the sum of the accumulator values $A_L + A_R \pmod M$.

Note that if all players behave, each client's output is

$$A_L + A_R \pmod M = \sum_i [x_i]_L^A +^p \sum_i [x_i]_R^A \pmod M = \sum_i x_i \pmod M$$

Since each x_i is at most 1, requiring $M > n$ ensures that $\sum_i x_i$ never overflows the modulus 2^l. This means that the output, interpreted as an integer, is indeed the sum $\sum_i x_i$.

The authors of [8] make two crucial observations about this simple scheme. First, it provides privacy as long as one server is honest. The adversary's view only includes a single share, say $[x]_L^A$, of an honest client's input x, which appears totally random without $[x]_R^A$. Second, the scheme does *not* provide robustness against malicious clients. A single malicious client can completely corrupt the protocol output by submitting (for example) arithmetic shares of a random integer $r \in \mathbf{Z}_M$ to each server.

The authors of Prio [8] solve the client robustness issue by forcing clients to construct and submit SNIPs (secret-shared non-interactive proofs), a novel type of zero-knowledge proof which allows the servers to non-interactively verify the form of client inputs. This is effective, but requires somewhat expensive computation and communication on the clients' part to construct and send these SNIPs to the servers. Since clients are presumed to be computationally weak compared to servers in our model, latency between clients and servers is high, this is not ideal. We would rather invoke some extra communication and computation among the servers if it would allow us to reduce the communication and computation on the client side.

We observe that clients can solve the robustness issue by instead submitting their data using the 1-bit Boolean secret-sharing scheme. This allows servers to verify that $x \in \{0, 1\}$ by simply confirming that each share is a single bit, forcing the client to sharing a value which is only a single bit in length. Then, servers can execute a share conversion protocol to privately convert these into arithmetic shares of the same secret values in the larger ring $\mathbf{Z}_M$ and sum them up as before. This

observation is the intellectual core of our deployment.

In the next section, we give further details of how we protect against malicious clients using share conversion. In particular, we show how to apply our observation described above to prevent clients from submitting data outside the range $\{0, 1\}$. This can be trivially generalized to verifying that client data falls within the range $[0, 2^l - 1]$. In future sections, we show how servers can verify other aspects of user inputs besides the bit-length. These tools will be necessary for computing more complex statistics. In that case, clients will encode their inputs before sharing with the servers, and servers will need tools to verify that submitted data is properly encoded. The authors of Prio [8] once again accomplish this via SNIPs, but we will show that such machinery is, in most cases, wasteful and unnecessary for computing the desired statistics.

6 Protecting Correctness via Share Conversion

In this section we give further details about share conversion and how we use it to make Prio+ robust against malicious clients.

The reason a malicious client can cheat and submit $x_i \notin \{0, 1\}$ is that when x_i is shared arithmetically in $\mathbf{Z}_M$, a single share reveals nothing about the size of x_i. From the servers' perspective, the underlying data x_i could be any element of $\mathbf{Z}_M$. Thus, server S_L holding a single share $[x_i]_L^A$ cannot tell whether $x_i \in \{0, 1\}$.

Imagine, however, that P_i shares x_i via the 1-bit Boolean scheme as $x_i = [x_i]_L^B \oplus [x_i]_R^B$. If $[x_i]_L^B \in \{0, 1\}$ and $[x_i]_R^B \in \{0, 1\}$, then it is guaranteed that $x_i \in \{0, 1\}$. Using Boolean to arithmetic share conversion, servers can then compute arithmetic shares $[x_i]^A$ of the value x_i in the extended ring $\mathbf{Z}_M$ and continue computation according to the simple scheme. If this conversion is done securely, these shares will have the same amount of entropy as the original 1-bit Boolean shares, effectively hiding the underlying secret value. Thus, all we need in order to make the simple scheme robust against malicious clients is a Boolean to arithmetic share conversion protocol achieving the following ideal functionality $\mathcal{F}_{B2A}$ described below.

Definition 6.1. The two-player l-bit ideal functionality $\mathcal{F}_{B2A}$ (with output in $\mathbf{Z}_M$) behaves as follows:

- $\mathcal{F}_{B2A}$ receives $[x]_L^B, [x]_R^B \in \mathbf{Z}_{2^l}$ as inputs from S_L, S_R respectively.
- $\mathcal{F}_{B2A}$ computes $x = [x]_L^B \oplus [x]_R^B$
- $\mathcal{F}_{B2A}$ computes $\mathsf{Share}_{+,M}(x) = ([x]_L^A, [x]_R^A)$ satisfying $[x]_L^A + [x]_R^A = x \pmod M$.
- $\mathcal{F}_{B2A}$ returns $[x]_L^A, [x]_R^A$ to S_L, S_R respectively as outputs.

Share conversion has been well-studied, both in its theoretical limitations [9] and its practical performance [14]. For this simple scheme, we will utilize the OT-based Boolean to Arithmetic share conversion protocol described in [14], as it has been shown to have the best practical performance given recent advancements in OT extension ([23], [29]). Their protocol utilizes l independent instances of correlated-OT, where each OT transfers an $(l + 1)/2$-bit string on average. This uses

$l(\kappa + (l+1)/2)$ total bits of communication. This is the most efficient technique due to modern advances in OT extension [23].

Later, when computing more advanced statistics, we will need the output of the share conversion to lie in a field $\mathbf{Z}_p$, which this OT-based share conversion cannot support. We will then use a simple but novel share conversion scheme based on extended doubly authenticated bits, or edaBits, which were first defined in [18]. It is known that edaBits can be utilized for share conversion, and this is quite common and well-studied, but they have been primarily used in the malicious setting, whereas we will use them to do share conversion with only semi-honest security. This can, of course, be done much more efficiently than in the malicious setting. For further details on edaBits and our share conversion protocol into $\mathbf{Z}_p$, please see Section 8.

From now on, we will use the notation $\mathsf{B2A}_{l,M}([x]_L^B, [x]_R^B)$ to represent an evaluation of a Boolean to arithmetic share conversion protocol with l-bit inputs and whose output lies in $\mathbf{Z}_M$. When the bitlength of the input shares and the output range are clear by context, we may simply write $\mathsf{B2A}([x]_L^B, [x]_R^B)$

With $\mathsf{B2A}$ in our toolbox, we can now strengthen the simple scheme from the previous section to prevent malicious clients from corrupting the output. In particular, we force clients to submit single bits by making them submit data under the 1-bit Boolean secret-sharing scheme. Then, servers convert the Boolean shares into the arithmetic scheme using $\mathsf{B2A}$. Servers then compute the sum of the shared data according to the simple scheme. Servers only accept shares consisting of a single bit, which means that if both servers accept shares $[x_i]_L^B, [x_i]_R^B$ from P_i, we are guaranteed that $[x_i]_L^B \oplus [x_i]_R^B \in \{0, 1\}$. This precisely guarantees robustness in the sense that a malicious client cannot affect the output of the protocol beyond misreporting their private data. A detailed description of this strengthened protocol can be found below. Detailed proofs of its privacy and robustness can be found in Appendix C.

Π_{bitSum}

Inputs: $x_i \in \{0,1\}$ for $i \in [n]$
Output: $\sum_{i=1}^{n} x_i$.

1. **Upload**:
   (a) Each client P_i computes $\mathsf{Share}_{\oplus,1}(x_i) \longrightarrow [x_i]_L^B, [x_i]_R^B$ via Definition 4.3
   (b) Each P_i sends $[x_i]_L, [x_i]_R$ to S_L, S_R respectively.
2. **Verify Bit-Length**: Initially, $n' = n$. If a server receives a share which is not 1 bit in length from P_i (assume S_L w.l.o.g.):
   (a) S_L sends the index i to S_R.
   (b) Both servers discard $[x_i]^B$.
   (c) Both servers set $n' \leftarrow n' - 1$
3. **Convert Shares**: S_L and S_R jointly evaluate $\mathsf{B2A}_{1,2^l}(\{[x_i]_L^B, [x_i]_R^B\})$ (as described in [14]) on each of the n' valid pairs of Boolean shares. S_L receives as output $\{[x_i]_L^A\}_i$ and S_R receives as output $\{[x_i]_R^A\}_i$.
4. **Aggregate**: S_L locally adds all arithmetic shares into an accumulator A_L, initially zero. That is: $A_L \longleftarrow A_L + \sum_i [x_i]_L^A$. S_R analogously accumulates its arithmetic shares into $A_R \longleftarrow A_R + \sum_i [x_i]_R^A$.
5. **Publish**: Once all n' shares have been accumulated, S_L and S_R publish A_L and A_R to every client.
6. **Client Computation**: Clients output $A_L + A_R$.

As we can see, this minor modification of the simple scheme guarantees both privacy and robustness without any heavy client-side computation. It also easily generalizes to compute the sum of l-bit integers for $l > 1$ by simply sharing data in the l-bit Boolean scheme and then using Boolean to arithmetic share conversion. If servers intentionally leak the number of honest clients n', then this protocol for computing the sum is sufficient for computing the arithmetic mean as well. The authors of Prio [8] extend their scheme beyond just computing the sum and arithmetic mean. Prio supports computation of a wide array of aggregation functions including: variance (VAR), standard deviation (STDDEV), Boolean OR and AND, integer MIN and MAX, frequency count (FRQ), and linear regression (linReg). They accomplish this by having users encode their input in particular ways such that the sum of the encoded inputs reveals the desired statistic. These are discussed in the literature as affine aggregatable encodings (AFEs), and we refer the reader to [8] for more information regarding these encodings.

The only obstacle in computing these statistics analogously using our system is to design a way for servers to successfully verify that clients' inputs are properly encoded. If this can be done, our Prio+ servers can verify that client data is properly encoded (and is within the proper range), perform share conversion, sum the resulting arithmetic shares and return the sum of the properly encoded inputs using the exact same technique as our Π_{bitSum} protocol. This sum of encoded inputs will be precisely the desired statistic according to the underlying AFE. Instead of universally relying on SNIPs for verifying client encodings, as Prio does [8], we use SNIPs sparingly. We use them exclusively for verifying multiplicative relationships within the encoding, and all other properties of the encoded inputs are verified via alternative, novel methods. We believe that this is a more appropriate application of SNIPs since they are, at their core, designed around verifying the outputs of multiplication gates within a validity circuit.

In the following section, we outline each of the more complex statistics that our scheme computes. For each statistic we describe the corresponding encoding for which summing the encoded inputs produces the desired statistic.

7 Complex Statistics

In this section, we describe each of the statistics our deployment computes. For each statistic, we describe the corresponding encoding which we use to enable computation. That is, how can we compute an encoding of x_i (written $\mathsf{en}(x_i)$) so that the statistic f we wish to compute is given by $f(x_1, \ldots, x_n) = \sum_i \mathsf{en}(x_i)$ (or some locally computable function of this sum). These encodings are referred to as "affine aggregatable encodings," or AFEs. As most of the machinery of AFEs is irrelevant to our applications, we omit a detailed discussion and instead refer readers to [8] for more information.

As a reminder, each client P_i holds input x_i from some secret-space $\mathcal{D}$ which will be encoded as $\mathsf{en}(x_i)$. They wish to compute $f(x_1, \ldots, x_n)$ using servers S_L, S_R. The servers are responsible for verifying that $\mathsf{en}(x_i)$ is a proper encoding of some $x_i \in \mathcal{D}$, as well as for summing these encodings and returning them to clients so they can reconstruct the value $f(x_1, \ldots, x_n)$. For each statistic f, we give the domain $\mathcal{D}$ of the input x_i (also referred to as the "secret-space") as well as the encoding

we will use for computing f. We will also give a brief intuition of why this encoding is sufficient and how the servers will use it to compute the statistic.

SUM, MEAN:

$\mathsf{SUM}(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$

$\mathsf{MEAN}(x_1, \ldots, x_n) = \frac{1}{n} \cdot \mathsf{SUM}(x_1, \ldots, x_n)$

Secret-Space: $\mathcal{D} = \mathbf{Z}_{2^l}$

Encoding: $\mathsf{en}_{\mathsf{int}}(x_i) = x_i$

Intuition: Clients submit their data secret-shared via the l-bit Boolean scheme. Servers use B2A to convert valid l-bit Boolean shares to arithmetic shares of the same secret x_i in $\mathbf{Z}_M$ for $M > n$ and then locally sum the resulting arithmetic shares. In order to compute the mean, we allow the servers to modestly leak the number of players $n - c$ whose valid shares are included in the aggregate. Then clients can locally compute the mean. Note: this means our integer mean protocol achieves only $\hat{f}$-privacy, where $\hat{f}$ leaks $n - c$.

AND, OR:

$\mathsf{AND}(x_1, \ldots, x_n) = 1 \iff \forall i, x_i = 1$

$\mathsf{OR}(x_1, \ldots, x_n) = 0 \iff \forall i, x_i = 0$

Secret Space: $\mathcal{D} = \{0, 1\}$

Encoding: $\mathsf{en}_{\mathsf{and}}(x_i) = (1 - x_i)\vec{r} \in \mathbf{F}_2^\lambda$ for some security parameter λ and random $\vec{r} \in \mathbf{F}_2^\lambda$. That is, if $x_i = 1$, $\mathsf{en}_{\mathsf{and}}(x_i) = \vec{0}$, and if $x_i = 0$, $\mathsf{en}_{\mathsf{and}}(x_i) = \vec{r}$.

$\mathsf{en}_{\mathsf{or}}(x_i) = x_i \cdot \vec{r} \in \mathbf{F}_2^\lambda$ for some security parameter λ and random $\vec{r} \in \mathbf{F}_2^\lambda$. That is, if $x_i = 0, \mathsf{en}_{\mathsf{or}}(x_i) = \vec{0}$, and if $x_i = 1, \mathsf{en}_{\mathsf{or}}(x_i) = \vec{r}$.

Intuition: Here, share conversion is unnecessary because the aggregation operator and the reconstruction operator for the Boolean secret-sharing scheme are both XOR. Thus, servers can simply locally XOR their valid shares and publish these aggregated values to clients, who will then XOR the aggregated values together to produce the output. When computing AND, if every client has $x_i = 1$, then every $\mathsf{en}(x_i) = \vec{0}$, and so the XOR of these encodings will certainly be $\vec{0}$. In this case, clients can conclude $\mathsf{AND}(x_1, \ldots, x_n) = 1$. Otherwise, if some client has $x_i = 0$, then $\mathsf{en}(x_i) = \vec{r}$ and the XOR of the encodings will be non-zero with probability $1 - \frac{1}{2^\lambda}$. In this case, they conclude that $\mathsf{AND}(x_1, \ldots, x_n) = 0$. The argument is analogous in the case of OR.

MAX, MIN:

$\mathsf{MAX}(x_1, \ldots, x_n) = \max_i x_i$

$\mathsf{MIN}(x_1, \ldots, x_n) = \min_i x_i$

Secret Space: $\mathcal{D} = \{0, \ldots, M\}$ for small $M \in \mathbf{Z}$

Encoding: $\mathsf{en}_{\mathsf{max}}(x_i) = (\vec{r}_1, \ldots, \vec{r}_i, \vec{0}, \ldots, \vec{0}) \in \mathbf{F}_2^{\lambda \times M}$, where each $\vec{r}_j \in \mathbf{F}_2^{\lambda}$ is independently random. This is equivalent to applying the $\mathsf{en}_{\mathsf{or}}()$ function to each component of the vector $(1, \ldots, 1, 0, \ldots, 0) \in \mathbf{F}_2^M$ where the first i components are 1.

Intuition: To compute the maximum, servers run the OR protocol M times in parallel on each component of the encoded input. That is, they analogously XOR their shares locally, and return them to clients to XOR and reconstruct the output. The clients parse this $(\lambda \times M)$-bit string in λ-bit chunks, reading each chunk as a 0 if and only if every bit of that chunk is 0. The clients compute the largest index k for which the corresponding OR protocol gave output 1, and conclude the maximum is k. That is, they compute the largest value k such that the k'th substring of λ consecutive bits contains a 1. This is certainly bounded above by the maximum, and the probability that it undershoots the maximum by Δ is $\frac{1}{2^{\lambda \times \Delta}}$, which is negligible in the security parameter λ.

To compute the minimum, clients first represent their input x_i as $\chi_i = M - x_i$, and compute the maximum of these χ_i values, $y_i = \mathsf{MAX}(\chi_1, \ldots, \chi_n)$, as above. The desired minimum will be precisely $\mathsf{MIN}(x_1, \ldots, x_n) = M - y_i$, with the same error bounds as the MAX protocol.

VAR, STDDEV:

$\mathsf{VAR}(x_1, \ldots, x_n) = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mathsf{MEAN}(x_1, \ldots, x_n))^2$

$\mathsf{STDDEV}(x_1, \ldots, x_n) = \sqrt{\mathsf{VAR}(x_1, \ldots, x_n)}$

Secret Space: $\mathcal{D} = \mathbf{Z}_{2^l}$

Encoding: $\mathsf{en}_{\mathsf{var}}(x_i) = (x_i, x_i^2)$

Intuition: Servers parse the encoded input into its two parts and compute shares of $(\sum_i x_i, \sum_i x_i^2)$ using two parallel instances of our protocol for SUM, which they return to clients. The clients divide these values by n, which is a public parameter, to compute $\mathbf{E}[X]$ and $\mathbf{E}[X^2]$, where X is a random variable taking on each value x_i with equal probability. From this, clients can locally compute $\mathsf{VAR}(x_1, \ldots, x_n) = \mathbf{E}[X^2] - (\mathbf{E}[X])^2$. Note: in the case where clients may misbehave, this protocol is only $\hat{f}$-private, where $\hat{f}$ leaks $\mathbf{E}[X]$ and the remaining number of behaving players n' in addition to the output. Clients who wish to compute the standard deviation simply add a local square root operation to the end of the protocol.

linReg:

$\mathsf{linReg}((x_1, y_1), \ldots, (x_n, y_n)) = (c_0, c_1)$, where $\hat{y}(x) = c_0 + c_1 x$ is the unique line which minimizes the sum of squares loss $\sum_i (y_i - \hat{y}(x_i))^2$.

Secret Space: $\mathcal{D} = \mathbf{Z}_{2^l} \times \mathbf{Z}_{2^l}$

Encoding: $\mathsf{en}_{\mathsf{reg}}(x_i, y_i) = (x_i, x_i^2, y_i, x_i y_i)$

Intuition: Analogously to the variance computation, servers compute the sum of the various parts of the encoding in parallel via our protocol for SUM and return shares of $(\sum_i x_i, \sum_i x_i^2, \sum_i y_i, \sum_i x_i y_i)$ to all clients. The clients can solve for the desired real regression coefficients c_0 and c_1 locally using the following linear system:

$$\begin{pmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{pmatrix} \cdot \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \end{pmatrix} \quad (1)$$

Note that again, in the case of misbehaving clients, this implies servers must also reveal the value $n - c$ of remaining players in the protocol, introducing a modest leakage. Thus this protocol will also be $\hat{f}$-private, where $\hat{f}$ leaks $n - c$ in addition to the output. This technique also trivially generalizes to d-dimensional client inputs $(x^{(0)}, x^{(1)}, \ldots, x^{(d)})$ for $d > 2$ as described by [8].

FRQ:

$\mathsf{FRQ}(x_1, \ldots, x_n) = \vec{h} = (f_1, \ldots, f_k) \in \mathbf{Z}_{n+1}^k$, where $f_j = |\{x_i : x_i = j\}| \leq n$ is the frequency of input $j \in \mathbf{Z}_k$

Secret Space: $\mathcal{D} = \{0, \ldots, k-1\}$ for small $k \in \mathbf{Z}$
Encoding: $\mathsf{en}_{\mathsf{frq}}(x_i) = (\delta_{x_i}) \in \mathbf{Z}_{2^k}$, where the x_i'th component of (δ_{x_i}) is 1 and all other components are 0. That is, (δ_{x_i}) is an impulse at x_i.
Intuition: If all players behave, taking the sum of these encodings yields the desired vector $\vec{h}$. Thus, servers evaluate k independent instances of B2A on each vector (one for each component) and then locally sum the resulting vectors of arithmetic shares. They then publish these aggregated vectors to clients who add them together to get $\vec{h}$ in the same manner as our SUM protocol.

We have now intuitively argued that, conditioned on the fact that all inputs are properly encoded, servers can efficiently and privately compute each of these desired statistics. All that remains is to describe the method by which servers can privately verify that shares $[x_i]_L, [x_i]_R$ reconstruct proper encoding of an element $x_i \in \mathcal{D}$ for each of the encodings described above.

In the case of SUM, MEAN, AND, OR, MAX, and MIN, we simply observe that any Boolean vector of the correct length is a valid encoding of some input in the relevant secret space. This makes our job simple, since servers can trivially verify the bit-length of the input by checking the bit-length of its shares.

For VAR, STDDEV, and linReg, servers must verify multiplicative relationships among different parts of the secret-shared encoded inputs. We accomplish this using SNIPs analogously to the methods of [8] described in Section 2. For a more detailed description of SNIPs construction and usage, see [8].

The trickiest encoding to verify is $\mathsf{en}_{\mathsf{frq}}$. Here, servers are given Boolean shares of v and they must verify that $\sum_i (v)_i = 1$. That is, v has exactly one component equal to 1. In other words, v is an impulse. This is a difficult encoding to verify because the number of valid encodings is quite small compared to the total number of proper-length Boolean vectors. This principle applies to all of our constructions: verifying an encoding is easiest when the set of proper encodings is "dense" within the set of all Boolean vectors of proper length. That is, if most or all Boolean vectors are proper encodings, little to no verification is necessary. In hindsight, this justifies the fact that verifying $\mathsf{en}_{\mathsf{VAR}}$ (and similarly linReg) requires so much additional machinery, since the density of

valid encodings in that case is $\frac{2^l}{2^{3l}} = \frac{1}{2^l}$.

In the case of FRQ, the proportion of valid encodings is very low, $\frac{l}{2^l}$. Furthermore, there does not seem to be any straightforward way to verify that a shared vector is a single impulse using only addition and multiplication of shares. Thus, our servers will have a bit more verification work to do. It would be desirable to construct some alternative encoding for computing frequency count which still permits computation but whose density of proper encodings in the set of Boolean vectors is higher. Despite these challenges, we give a novel and practically efficient technique for verifying this encoding in the next section.

8 Resolving Frequency Count

In this section we give a unique method for servers S_L, S_R, given $[x]^B \in \mathbf{Z}_{2^l}$, to verify that x is an impulse. In particular, we detail how to do this efficiently in parallel for $[x_1]^B, \ldots, [x_n]^B$. When we say x is an impulse, we mean $(x)_j = 1 \iff j = j^*$ for some unique $j^* \in \{0, \ldots, k-1\}$. Crucially, they must do this without leaking anything about honest players' input x_i besides whether or not it is an impulse.

Intuitively, we break this process down into two parts:

1. Make sure no player submitted a zero vector.
2. Make sure no player submitted multiple impulses by verifying that the sum of components in the final histogram equals n', the number of valid shares received.

We can perform the first check on all n inputs using just $2n$ bits of communication and $n \cdot l$ XOR operations. This is accomplished by checking the parity of the number of ones in each vector and throwing out any vector with an even number of ones. Servers can perform this parity check in the clear since it will never reveal any information about an honest player's input besides the fact that there are an odd number of ones, which is already implied by the fact that they submitted an honest impulse vector. Thus, each server simply computes the parity of their share and they combine these parities in the clear to get the parity of the client's encoded input. This uses exactly two bits of communication and l XOR operations per server for a single parity check. More precisely:

- S_L, S_R initially hold $[x]^B_L$, $[x]^B_R$ respectively.
- S_L computes $b_L = \bigoplus_j ([x]^B_L)_j$ and sends b_L to S_R.
- S_R computes $b_R = \bigoplus_j ([x]^B_R)_j$ and sends b_R to S_L.
- Both servers compute $b = b_L \oplus b_R$.
- If $b = 0$, both servers discard $[x]^B$.

Now, assume the first condition holds. Since no player submitted a zero vector, we must now detect players who submitted "bad" vectors with multiple ones in them. We know from the first condition that if the sum of the components in the final histogram, which we call $\mathsf{sum}(\vec{h})$, is precisely n', the

number of players who submitted non-zero vectors, we can conclude that no player submitted a bad vector of multiple ones.

We would ideally like to allow servers to simply sum the components of their local shares, producing a single arithmetic share of $\mathsf{sum}(\vec{h})$. Then, they could reconstruct this value and compare it to n'. This would leak no information about honest players' inputs, since it is already known that honest players submitted a single impulse, and this sum reveals nothing about the location of any particular impulse, only the total number of such impulses. Since components of the input vectors are originally shared using the Boolean scheme, however, we must first convert each component into the arithmetic secret-sharing scheme using $\mathsf{B2A}$. Once we make this conversion, servers can sum these vectors of Arithmetic shares locally to produce arithmetic shares of $\mathsf{sum}(\vec{h})$, reconstruct this in the clear, and compare it to n', as detailed above.

In the best case, if this check succeeds the first time we attempt it, we know with certainty that no player submitted a non-impulse vector (conditioned on the fact that no player submitted a zero vector, as confirmed by the initial verification step). If it fails, we split the set of behaving players in half and recursively repeat this check on each half, locating misbehaving players via binary search. In the worst case, this brings the total communication to $2nl \cdot \log(n)$ bits of communication and $O(n \log n)$ local additions/multiplications.

As we mentioned, the rest of the verification is simple: servers compute $s_L = \sum_i ([x]^*_L)_i$ and $s_R = \sum_i ([x]^*_R)_i$. We will then have that $s_L + s_R = \mathsf{sum}(\vec{h})$. To compare this value to n', S_L sends s_L to S_R and S_R computes $s_L + s_R - n'$. If this value is zero, then all inputs are single impulses and S_R replies with the bit 1. If not, S_R replies with the bit 0, indicating that some input has failed, and they continue the recursive search for misbehaving clients.

Once both verification steps succeed, we know that every player submitted a valid impulse except with negligible probability. From this point, servers proceed to compute the frequency count by locally summing their valid vectors of arithmetic shares into local accumulators $A_L, A_R \in \mathbf{Z}^k_{2^l}$. If each such vector is valid, then $A_L + A_R = \vec{h}$, the desired histogram. Servers then return A_L, A_R to clients who reconstruct $\vec{h}$ in precisely this manner.

9 Share Conversion

In this section, we detail our Boolean-to-arithmetic share conversion protocol based on edaBits [18]. As opposed to the OT-based protocol from [14], the arithmetic shares output by our protocol lie in the field $\mathbf{Z}_p$. This is necessary for our protocols which utilize SNIPs (variance, linReg) because the arithmetic shares must be used within a polynomial identity testing procedure. Such a procedure is not valid in a ring, where a polynomial of degree d may have arbitrarily many zeroes.

First, we review the definition and construction of edaBits. An l-bit edaBit consists of the following: $([r]^A, [r_0]^B, \ldots, [r_l]^B)$ where $r_i \in \{0, 1\}$ and $r = \sum_{i=1}^{l} r_i \cdot 2^i$ is an l-bit integer and $[r]^A$ is shared in $\mathbf{Z}_p$ for some $p > 2^l$. This is a refinement of the daBit primitive, originally introduced by [31]. A daBit is simply a random bit b shared in both the arithmetic and Boolean schemes. That is, $([b]^A, [b]^B)$ for $b \in \{0, 1\}$. Their paper, as well as other followup papers including [2], show how to

use l daBits to convert Boolean shares of an l-bit integer into a random arithmetic sharing of the same integer. In [18], the fact that these daBits can be replaced by a single edaBit is discussed but the resulting protocol is not made explicit.

Our edaBit generation protocol is identical to the edaBit generation protocol from [18]. Recall, the goal of edaBit generation is for k players, each with access to their own independent randomness, to compute a shared vector $([r]^A, [r_0]^B, \ldots, [r_b]^B)$ where r is a b-bit random value unknown to all players and $r = \sum_i 2^i r_i$. In the semi-honest setting, the procedure for generating "faulty" edaBits (edaBits which may or may not be properly formed) given in [18] always results in proper edaBits since our servers follow the protocol honestly. The basic idea is that each party generates their own private edaBit, with underlying values r, r', and sends one half of the shares to the other party. Then, they combine the private edaBits into a public edaBit whose underlying value $r + r'$ is unknown to both parties. This can be done trivially for the arithmetic shares by simply computing $[r]^A + [r']^A = [r + r']^A$. For the Boolean shares, players pass the bits r_i, r'_i through an l-bit binary adder circuit composed of AND gates and XOR gates. All gates of the form $x \oplus y$ can be trivially evaluated by locally computing $[x \oplus y]^B = [x]^B \oplus [y]^B$. Since AND is equivalent to single-bit multiplication, each AND gate can each be evaluated at the cost of two oblivious transfers (OTs). The naive l-bit binary adder circuit consists of $l + 1$ AND gates. Thus, to generate an l-bit edaBit requires $2(l + 1)$ OTs. To ensure that the value $r + r'$ is an l-bit integer, servers use a single daBit to convert their Boolean shares of the final carry bit c from the binary adder circuit into arithmetic shares $[c]^A$. The final result of edaBit generation is the arithmetic shares $[r + r' - 2^n \cdot c]$, which is an l-bit integer unknown to both parties, and Boolean shares of those l bits $[r_0]^B, \ldots, [r_l]^B$ representing $r + r' - 2^n \cdot c$. More details can be found in [18].

Our method of using edaBits for share conversion is strikingly similar to the generation procedure. Given $([x_0]^B, \ldots, [x_l]^B)$ and $([r]^A, [r_0]^B, \ldots, [r_l]^B)$, we wish to compute $[x]^A$, where $x = \sum_{i=0}^{l} x_i \cdot 2^i$. Note that if we are able to compute $[x + r]^A$, we can then trivially compute $[x]^A = [x + r]^A - [r]^A$. To compute $[x + r]^A$, we again utilize the binary adder circuit strategy detailed above, this time with inputs $([x_0]^B, \ldots, [x_l]^B), ([r_0]^B, \ldots, [r_l]^B)$. The key difference is that our output should be arithmetic shares $[x+r]^A$ in $\mathbf{Z}_p$, rather than a list $([s_0]^B, \ldots, [s_l]^B)$ of l Boolean shares representing the bits s_i of the sum $s = x + r$. Since x and r are both unknown to both parties, $x + r$ can be revealed in the clear without compromising privacy. Thus parties simply reconstruct $x + r$ in the clear, recompute an arithmetic sharing $[x + r]^A$, and compute $[x]^A = [x + r]^A - [r]^A$ as desired. Formal descriptions of both the edaBit generation protocol and the edaBit-based share conversion protocol follow.

edaBitGen

Inputs: One daBit $([d]^A, [d]^B)$ for random $d \in \{0,1\}$
Output: An l-bit edaBit $([r]^A, [r_0]^B, \ldots, [r_l]^B)$

1. **Sample Private edaBits**:
   (a) S_L samples random $r \in \mathbf{Z}_{2^l}$ and computes $([r]^A, [r_0]^B, \ldots, [r_l]^B)$, where $r = \sum 2^i \cdot r_i$. Similarly, S_R samples random $r' \in \mathbf{Z}_{2^l}$ and computes $([r']^A, [r'_0]^B, \ldots, [r'_l]^B)$, where $r' = \sum 2^i \cdot r'_i$.
   (b) S_L sends $([r]^A_R, [r_0]^B_R, \ldots, [r_l]^B_R)$ to S_R, and S_R sends $([r']^A_L, [r'_0]^B_L, \ldots, [r'_l]^B_L)$ to S_L.
2. **Combine Arithmetic Shares**:
   (a) S_L locally computes $[r + r']^A_L = [r]^A_L + [r']^A_L$.
   (b) S_R locally computes $[r + r']^A_R = [r]^A_R + [r']^A_R$.
3. **Combine Boolean Shares**:
   (a) Using an l-bit binary adder circuit over secret-shared values composed of AND gates and XOR gates, compute $([(r + r')_0]^B, \ldots, [(r + r')_l]^B)$, where $(r + r')_i$ is the i'th bit of $r + r'$, as well as $[c]^B$, a Boolean share of the carry bit $c \in \{0,1\}$.
      i. The first l-bit input to the circuit will be $r_0, \ldots, r_l$ and the second will be $r'_0, \ldots, r'_l$.
      ii. To compute an XOR gate $b_i \oplus b_j$ given $[b_i]^B$ and $[b_j]^B$, players locally compute $[b_i \oplus b_j]^B = [b_i]^B \oplus [b_j]^B$.
      iii. To compute an AND gate $b_i \cdot b_j$ given $[b_i]^B$ and $[b_j]^B$, players use two instances of 1-2-OT. Suppose P_1 holds $[b_i]_1$ and $[b_j]_1$, P_2 holds $[b_i]_2$, $[b_j]_2$. $b_i \cdot b_j = ([b_i]_1 \oplus [b_i]_2) \cdot ([b_j]_1] \oplus [b_j]_2) = [b_i]_1 \cdot [b_j]_1 \oplus [b_i]_1 \cdot [b_j]_2 \oplus [b_i]_2 \cdot [b_j]_1 \oplus [b_i]_2 \cdot [b_j]_2$. P_1's share of $b_i \cdot b_j$ will be $([b_i]_1 \cdot [b_j]_1) \oplus ([b_i]_1 \cdot [b_j]_2)$. The first part is computed locally. For the cross-term, P_1 acts as the receiver in a 1-2-OT with choice bit $[b_i]_1$. P_2 acts as the sender with messages $m_0 = 0 \cdot [b_j]_2 = 0$ and $m_1 = 1 \cdot [b_j]_2 = [b_j]_2$. Thus P_1 receives $[b_i]_1 \cdot [b_j]_2$. P_2's share $[b_i]_2 \cdot [b_j]_1 \oplus [b_i]_2 \cdot [b_j]_2$ is computed symmetrically, using another 1-2-OT instance.
      iv. Output: $([(r + r')_0]^B, \ldots, [(r + r')_l]^B)$ and $[c]^B$
   (b) To account for the carry operation, servers evaluate B2A from [18] on $[c]^B$ to get $[c]^A$ (cost: one daBit). They then locally compute $[r + r' - 2^n \cdot c]^A \leftarrow [r + r']^A - 2^n \cdot [c]^A$. The resulting value $r + r' - 2^n \cdot c$ is the l-bit integer represented by $(r + r')_0, \ldots, (r + r')_l$.
4. **Output**: $([r + r' - 2^n \cdot c]^A, [(r + r')_0]^B, \ldots, [(r + r')_l]^B)$

$\mathsf{B2A}_{l,p}$

Inputs: $[x]_L^B, [x]_R^B \in \mathbf{Z}_2^l$, $([r]^A, [r_0]^B, \ldots, [r_l]^B)$ from edaBitGen, $l+1$ Boolean multiplication triples $\{[a_i], [b_i], [a_i b_i]\}$ for $i \in [l+1]$
Output: $[x]_L^A, [x]_R^A \in \mathbf{Z}_p$.

1. **Add Boolean Shares via Binary Adder Circuit**:
   (a) Using an l-bit binary adder circuit over secret-shared values composed of AND gates and XOR gates, compute $([s_0]^B, \ldots, [s_l]^B)$, where s_i is the i'th bit of $s = x + r$.
      i. The first l-bit input to the circuit will be $r_0, \ldots, r_l$ and the second will be $x_0, \ldots, x_l$.
      ii. To compute an XOR gate $b_i \oplus b_j$ given $[b_i]^B$ and $[b_j]^B$, players locally compute $[b_i \oplus b_j]^B = [b_i]^B \oplus [b_j]^B$.
      iii. To compute an AND gate $b_i \cdot b_j$ given $[b_i]^B$ and $[b_j]^B$, players use two instances of 1-2-OT as described in edaBitGen.
2. **Compute** $[x]^A$:
   (a) Send all shares $[s_i]^B$ to a single participant P to reconstruct the value $x + r = \sum_i 2^i \cdot s_i$.
   (b) P computes a random Arithmetic sharing $[x + r]^A$ and shares this with the other participant(s).
   (c) All parties compute $[x]^A = [x + r]^A - [r]^A$.
3. **Output**: $[x]^A$

Communication Complexity: Both the generation and share conversion procedure have a communication complexity of $O(l + \log p)$.

10 Security

In this section we briefly describe the security properties of the protocols in our system. For formal statements and proofs, see Appendix C.

Let $0 \leq c^* \leq n$ be the number of corrupted clients who submit invalid input shares. For every protocol, up to n malicious players colluding with one semi-honest server learn nothing but the output except with negligible probability, as well as some modest leakage in some cases based on the specific statistic and/or AFE construction. In particular, to compute MEAN servers must leak the number of players $n - c^*$ whose inputs were included in the aggregate. Servers must also give this value when computing linReg. Due to the specific AFE construction for VAR, the output necessarily leaks $\mathbf{E}[X]$ in addition to the variance. All of these modest leakages are analogous to the results of [8].

In terms of robustness, all protocols provably prevent any coalition of up to n malicious players from corrupting the output beyond misreporting their private values, except with negligible probability. All statements given above are true, as proven in Appendix C, except with negligible probability.

11 Practical Evaluation

In this section we describe the practical performance of our implementation of this data-collection scheme.

We implemented our scheme in 7,000 lines of C/C++. We utilized the libOTe toolkit [30] for OT and silent OT-extension. Our scheme uses semi-honest OTs, since our servers are assumed to be semi-honest. Similar to Prio, clients use NaCl's "box" primitive to encrypt and sign messages. This means that TLS is not required to secure client-server communication.

Our implementation supports secure computation of SUM, AND, OR, MAX, MIN, VAR, FRQ and linReg. Two implementations of the original Prio protocol exist: the original implementation, written in Go, supports secure computation of SUM, AND, OR, MAX, MIN, and linReg. Since the original paper's publication, another implementation was written by Mozilla in C. The Mozilla implementation only supports SUM.

We provide comparison data for three statistics: SUM, MAX, and linReg. These represent our three categories of protocols: SUM requires share conversion but no SNIPs, MAX requires neither share conversion nor SNIPs, and linReg requires both share conversion and SNIPs.

We collected four types of data for comparison: client encode time (milliseconds per client), client message size (bytes per client per server), server compute time (milliseconds per server per client), and server communication (bytes per server per client). Most data we collected shows a direct linear relationship with the total number of clients, and so we ran multiple trials with 10k, 50k, and 100k clients and then averaged the result. The only exception to this is server compute time for VAR and linReg. Since they utilize SNIPs which are easily batched, server compute time increases sublinearly with number of clients. In that case only, we performed trials with 50k clients and averaged those results. In our end-to-end implementation we used the exact same servers as the original Prio paper, two c4.2x large AWS servers. All protocols were implemented using just two servers. Both servers are located in the us-east-1f zone as to mimic a low-latency, high bandwidth connection. The client code was run from a separate instance of the same c4.2x large AWS server, and all client data was randomly generated. All three implementations (Go, Mozilla, Prio+) use this same 2-server setup.

11.1 Data: SUM

For SUM, we compared Prio+ to both the original Go implementation and the Mozilla implementation. We ran four separate experiments with clients holding 1-bit, 8-bit, 16-bit, and 32-bit integers. Our results can be found in Figures 1, 2, 3, and 4. We also measured end-to-end runtime of our system. In total, our Prio+ implementation computes the sum of 100,000 16-bit integers in 0.47 seconds. This excludes offline pre-computation time but includes client encode time, communication time, and server compute time.

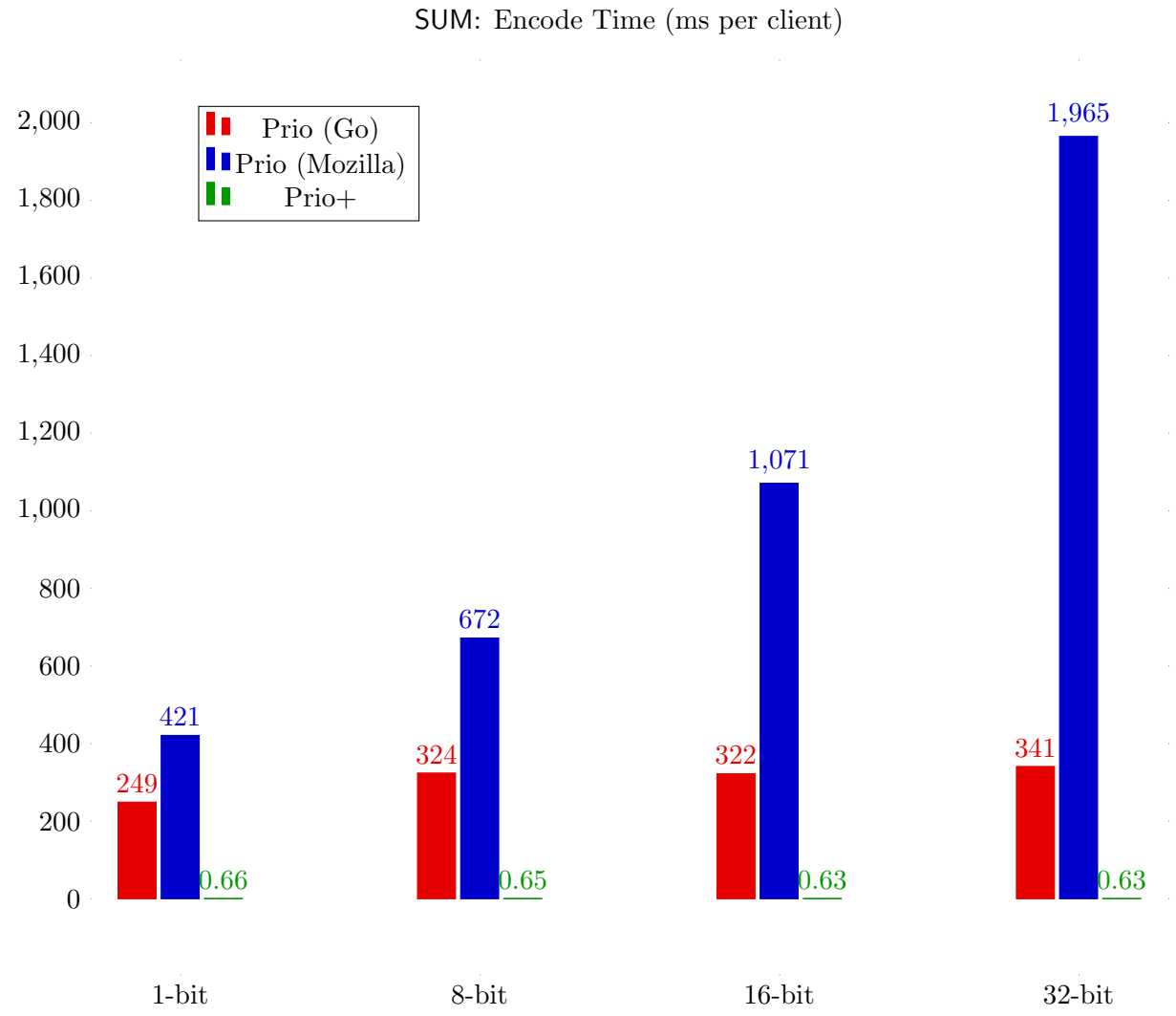

Figure 1: This chart shows the time necessary for a single client P_i holding private value x_i to encode that value, compute any necessary additional proofs, and secret-share both the encoding and the proof(s) when executing a protocol to compute $\mathsf{SUM}(x_1, \ldots, x_n)$. Results are in milliseconds and data is arranged according to the number of bits in x_i.

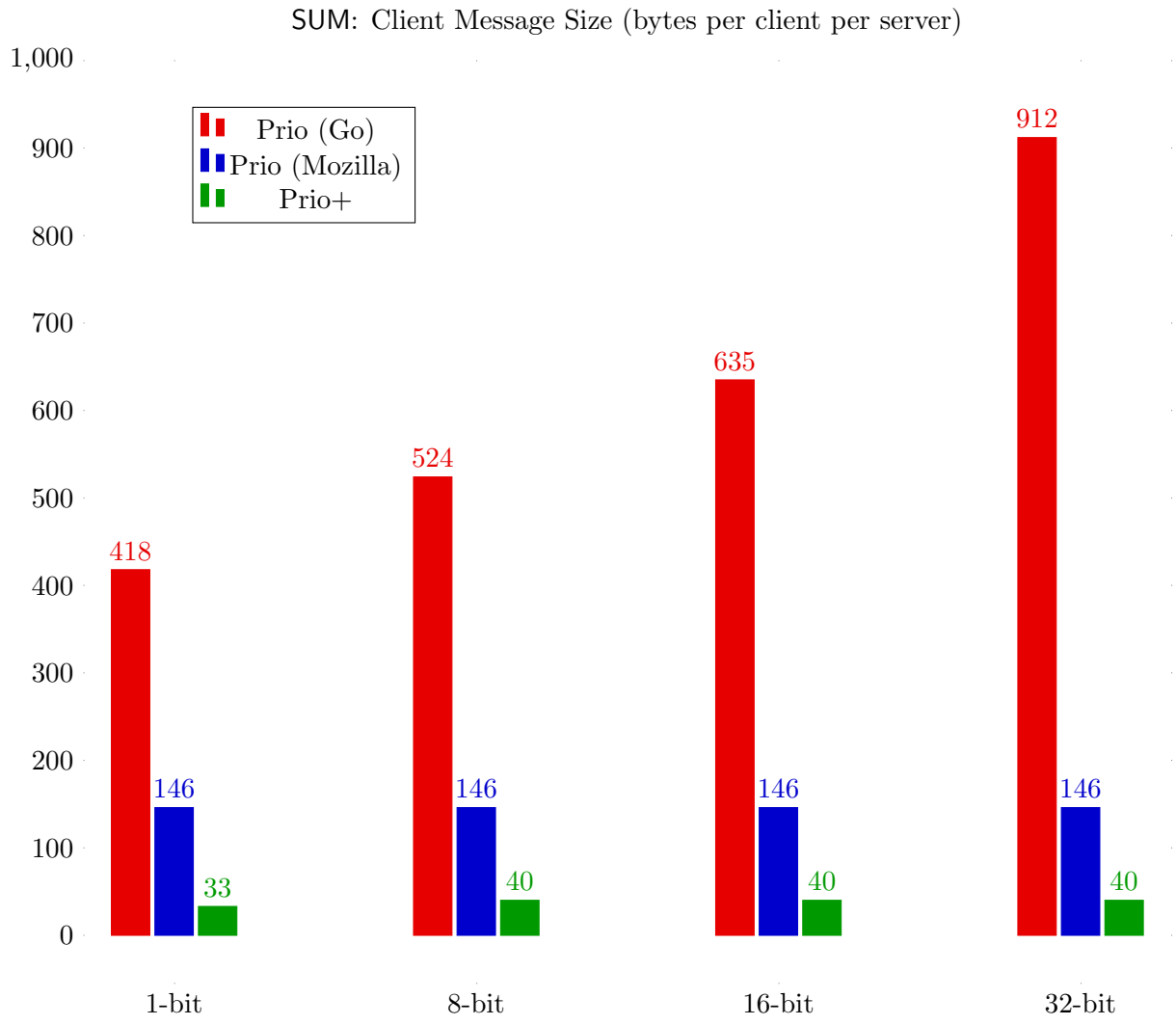

Figure 2: This chart shows the size of the message P_i (holding private value x_i) sends to each server when executing a protocol to compute $\mathsf{SUM}(x_1, \ldots, x_n)$. Results are in bytes and data is arranged according to the number of bits in x_i.

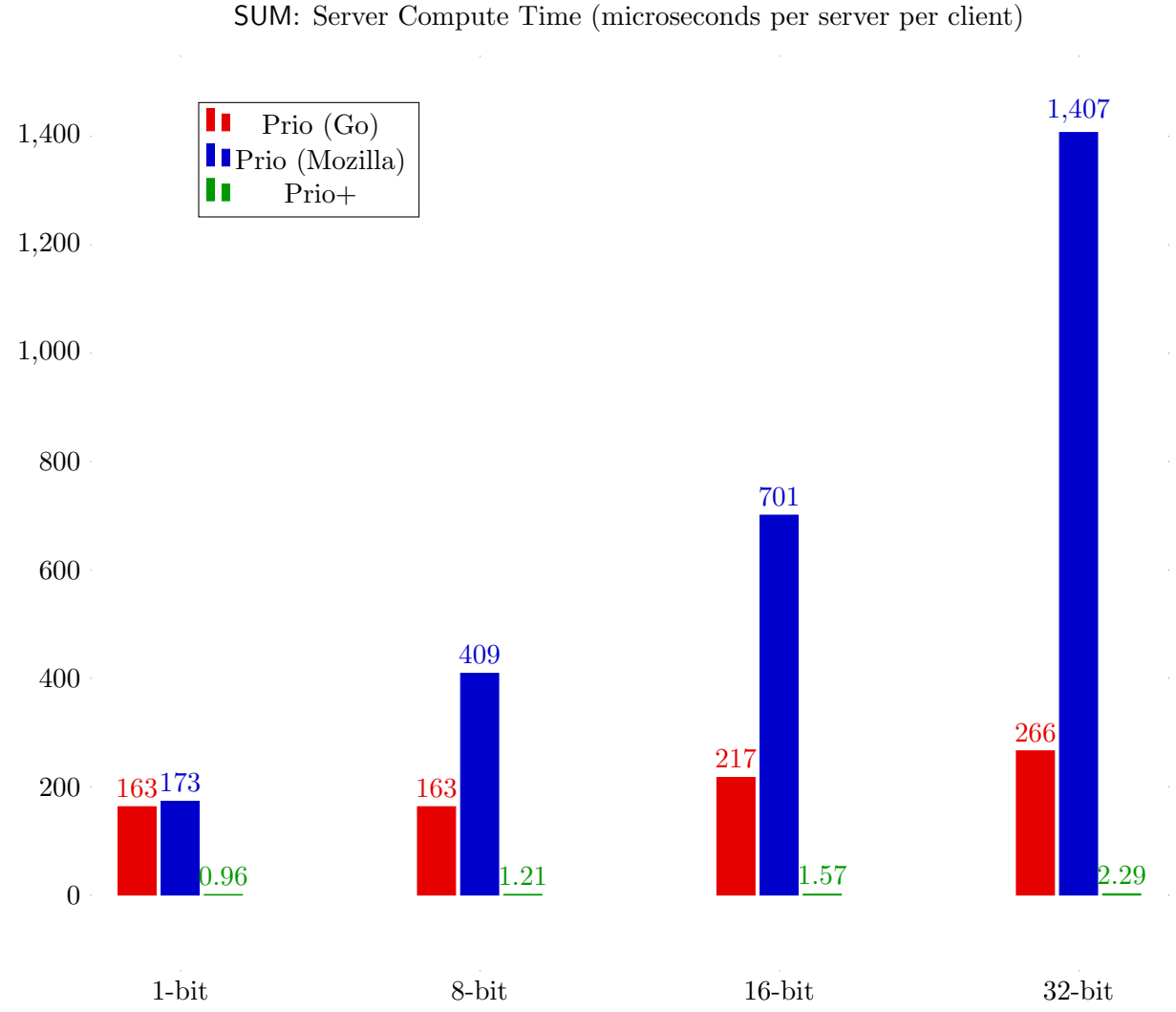

Figure 3: This chart shows the average computation time per server when executing a protocol to compute $\mathsf{SUM}(x_1, \ldots, x_n)$. Results are in milliseconds and data is arranged according to the number of bits in x_i, the private values held by each client.

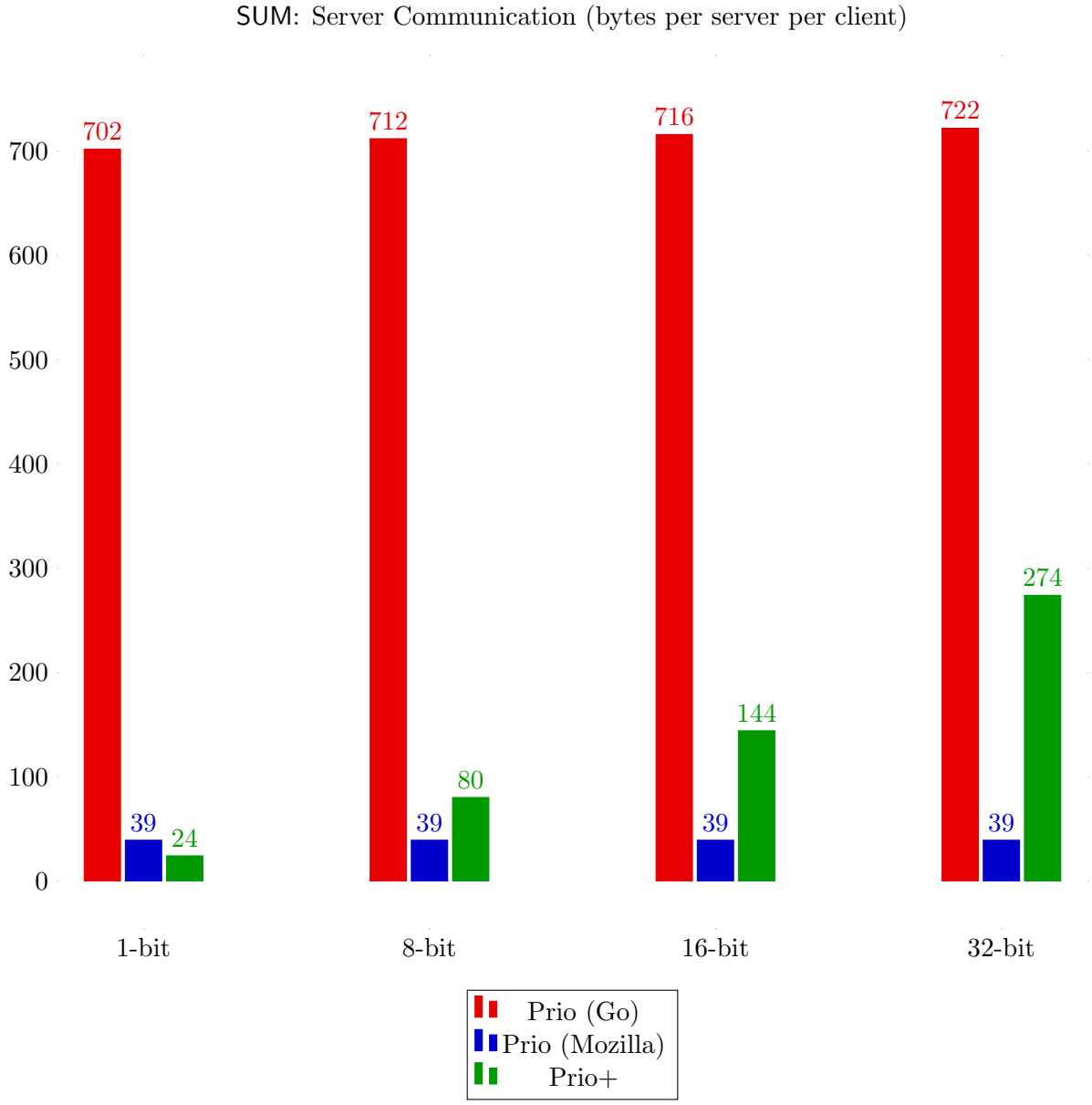

Figure 4: This chart shows the average bytes communicated by each server when executing a protocol to compute $\mathsf{SUM}(x_1, \ldots, x_n)$. Results are in bytes and data is arranged according to the number of bits in x_i, the private values held by each client.

Analysis: As expected, our implementation of Prio+ overwhelmingly outperforms both implementations of Prio in terms of client encode time and client message size. An additional pleasant result is that Prio+ heavily outperforms the Prio implementations in terms of server compute time as well. We expected to see performance drawbacks in server communication, since our server communication is no longer constant with respect to input size. Prio+ has comparable server communication to the Mozilla implementation for summing single-bit integers, but as the size of the integers increases, so does the communication. The Go implementation has high, but still constant, server communication. This is due to it being designed for a larger number of servers, whereas our experiments were run on a two-server implementation. Prio+ servers still communicate less than those in the original Go implementation except in the 32-bit case. The Mozilla implementation achieves lower server communication than Prio+, and this is the expected drawback of using share conversion as a replacement for SNIPs, which were designed to minimize server communication only.

11.2 Data: MAX

Since the Mozilla implementation does not support MAX, we compared Prio+ to the original Go implementation only. In our experiment, clients held integers in the range $[0, x]$ for $x \in \{16, 32, 64, 128\}$. In total, our Prio+ implementation computes the maximum of 100,000 4-bit integers in 5.25 seconds. This excludes offline pre-computation time but includes client encode time, communication time, and server compute time.

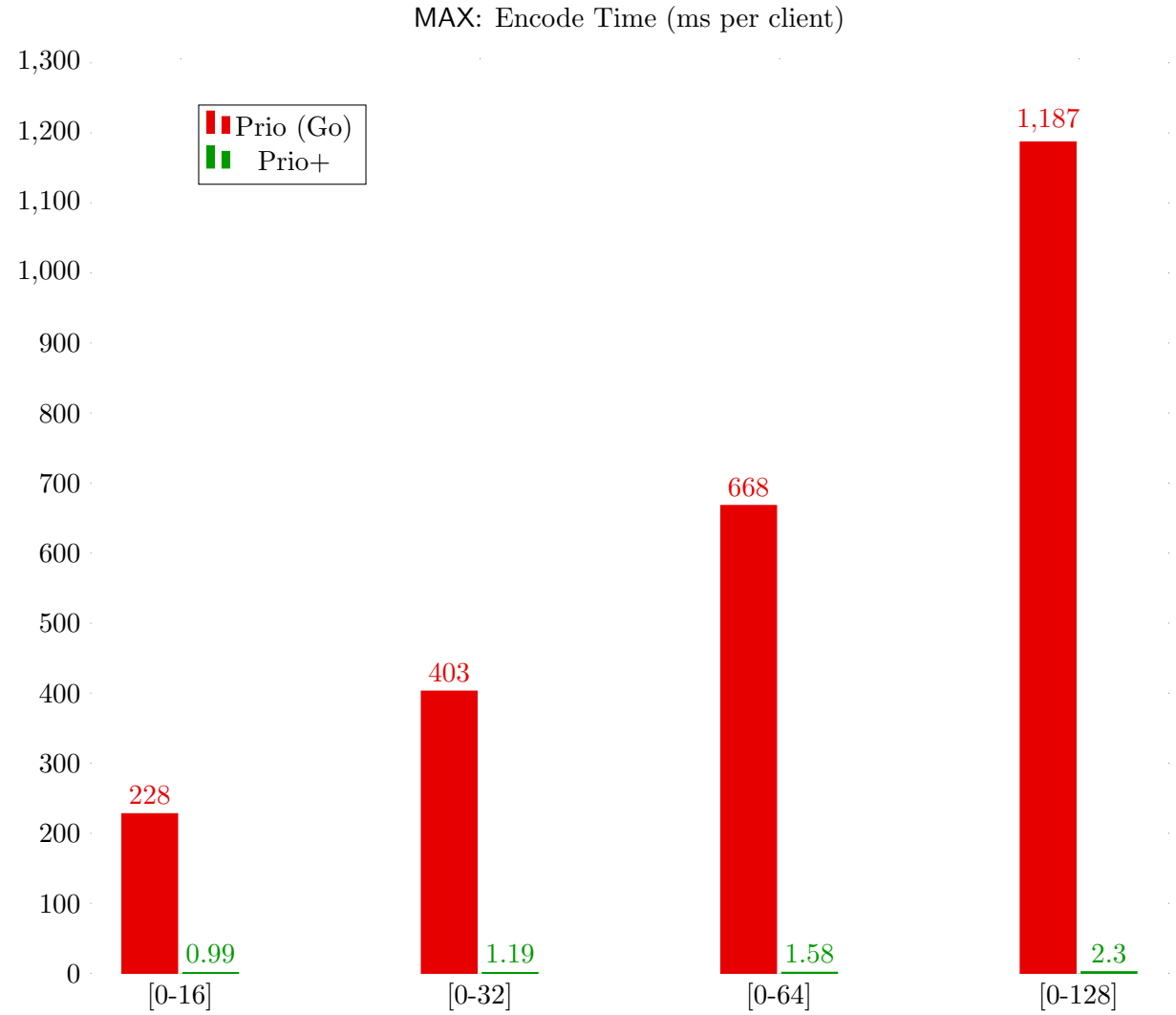

Figure 5: This chart shows the time necessary for a single client P_i holding private value x_i in the range $[0, x]$ for $x \in \{16, 32, 64, 128\}$ to encode that value, compute any necessary additional proofs, and secret-share both the encoding and the proof(s) when executing a protocol to compute $\mathsf{MAX}(x_1, \ldots, x_n)$. Results are in milliseconds.

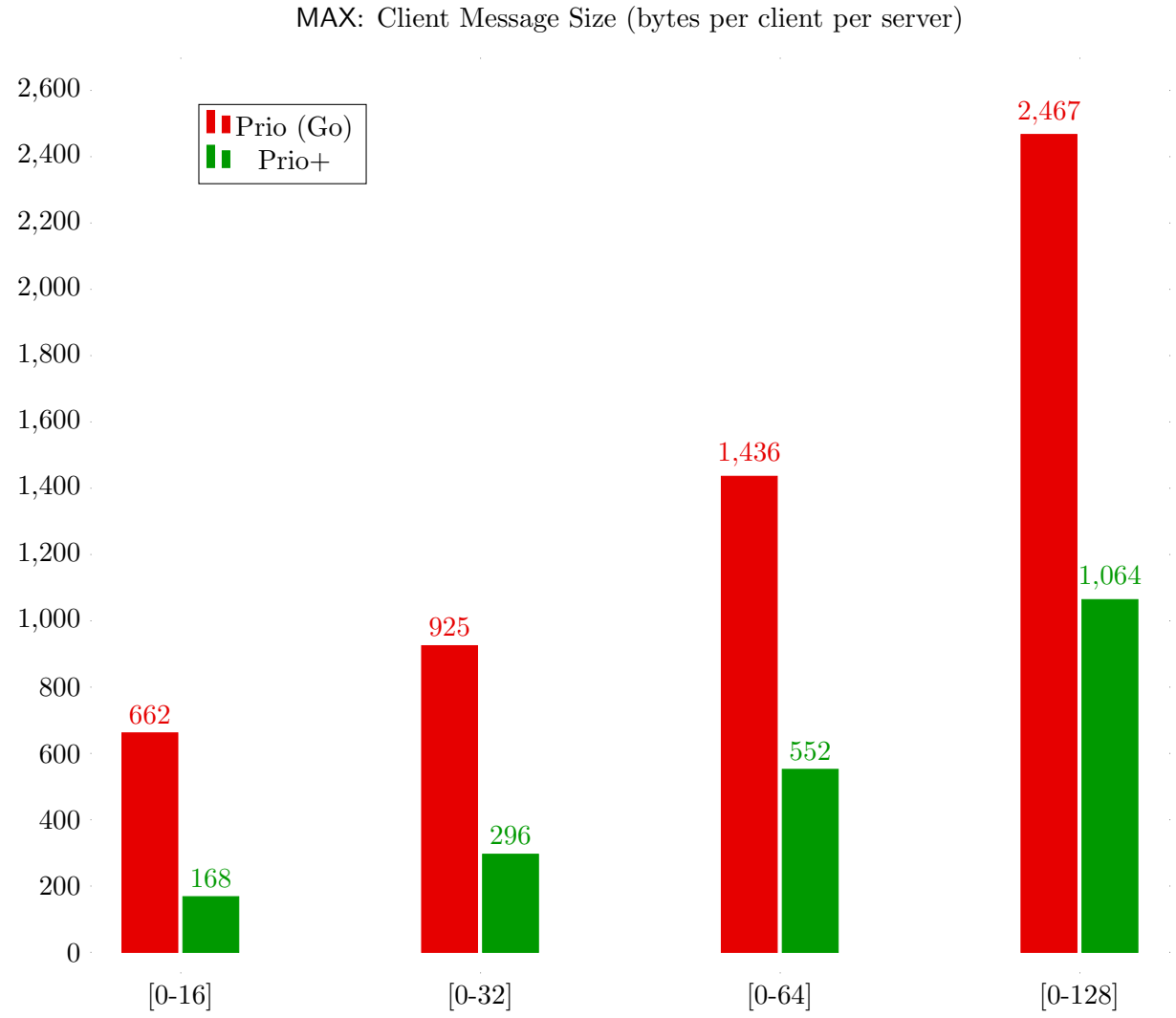

Figure 6: This chart shows the size of the message P_i (holding private value x_i in the range $[0, x]$ for $x \in \{16, 32, 64, 128\}$) sends to each server when executing a protocol to compute $\mathsf{MAX}(x_1, \ldots, x_n)$. Results are in bytes.

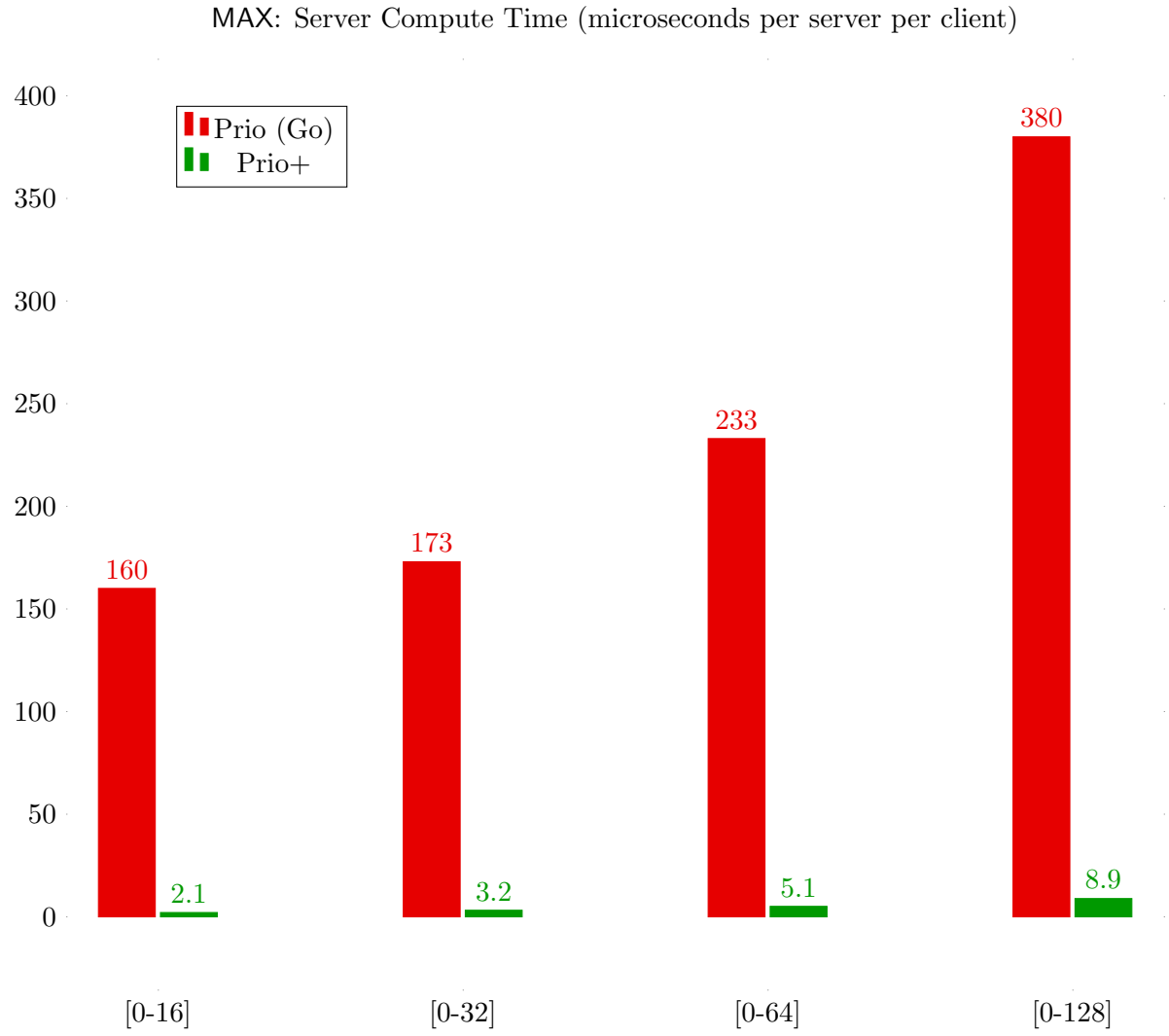

Figure 7: This chart shows the average computation time per server when executing a protocol to compute $\mathsf{MAX}(x_1, \ldots, x_n)$, where each x_i held by P_i lies in the range $[0, x]$ for $x \in \{16, 32, 64, 128\}$. Results are in milliseconds.

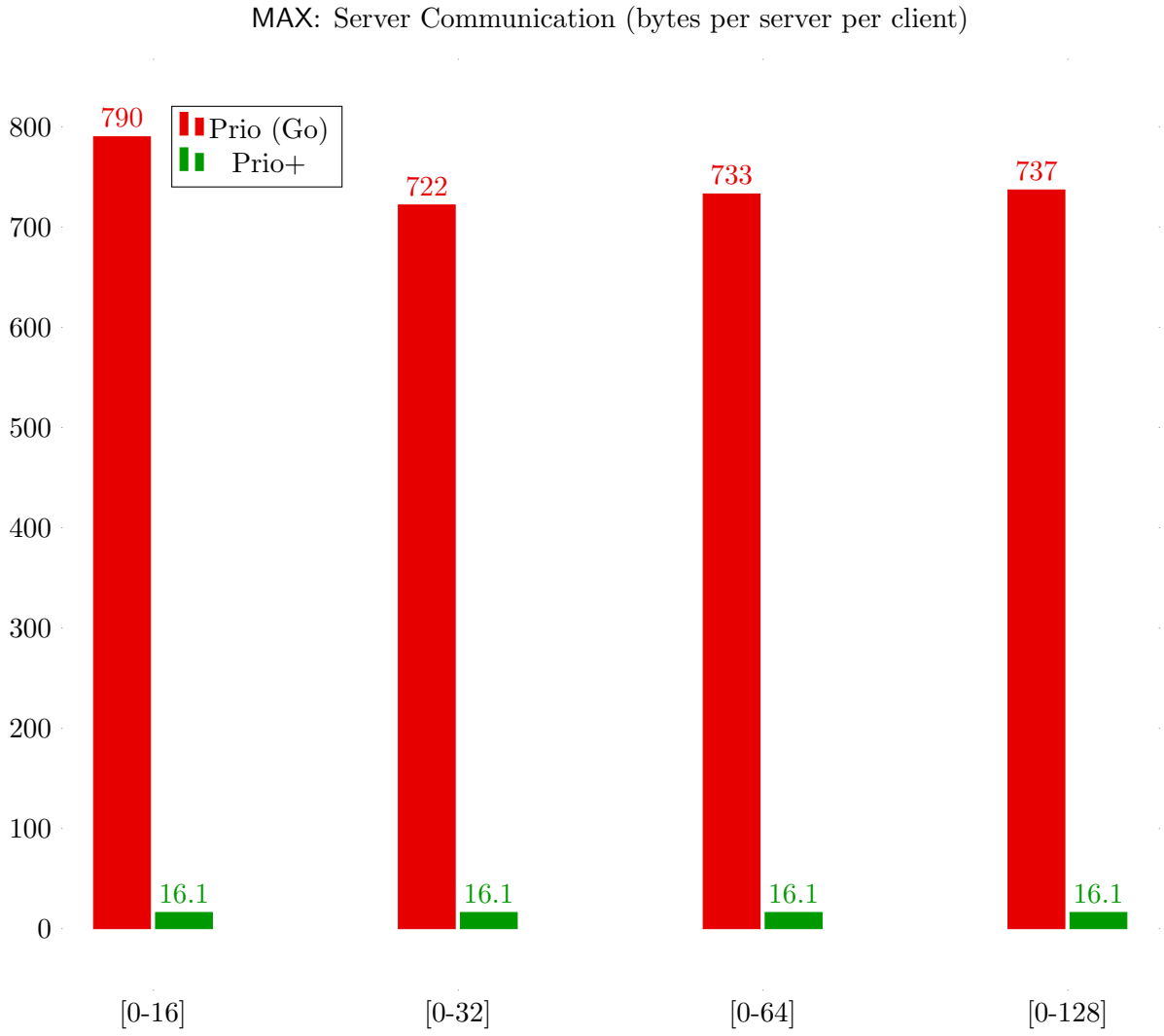

Figure 8: This chart shows the average bytes communicated by each server when executing a protocol to compute $\mathsf{MAX}(x_1, \ldots, x_n)$, where each x_i held by P_i is in the range $[0, x]$ for $x \in \{16, 32, 64, 128\}$. Results are in bytes.

Analysis: In the case of MAX, Prio+ requires no share conversion and no SNIPs, resulting in dramatic performance benefits for both clients and servers. Particularly relevant is the nearly 1,000x decrease in client encode time. Since servers do not have to perform share conversion, we get an added benefit of sharply decreased server communication.

11.3 Data: linReg

Similar to MAX, linReg is only supported by Prio+ and the original Go implementation. We performed four separate experiments in which the feature vectors held by clients are of degree 2, 4, 6, and 8 respectively. All results are averaged between experiments with 10k, 50k and 100k clients except for server compute time where all experiments were performed with 50k clients. In total, our Prio+ implementation computes a line-of-best-fit over 100,000 degree-2 vectors of 8-bit integers in 7.41 seconds. This excludes offline pre-computation time but includes client encode time, communication time, and server compute time.

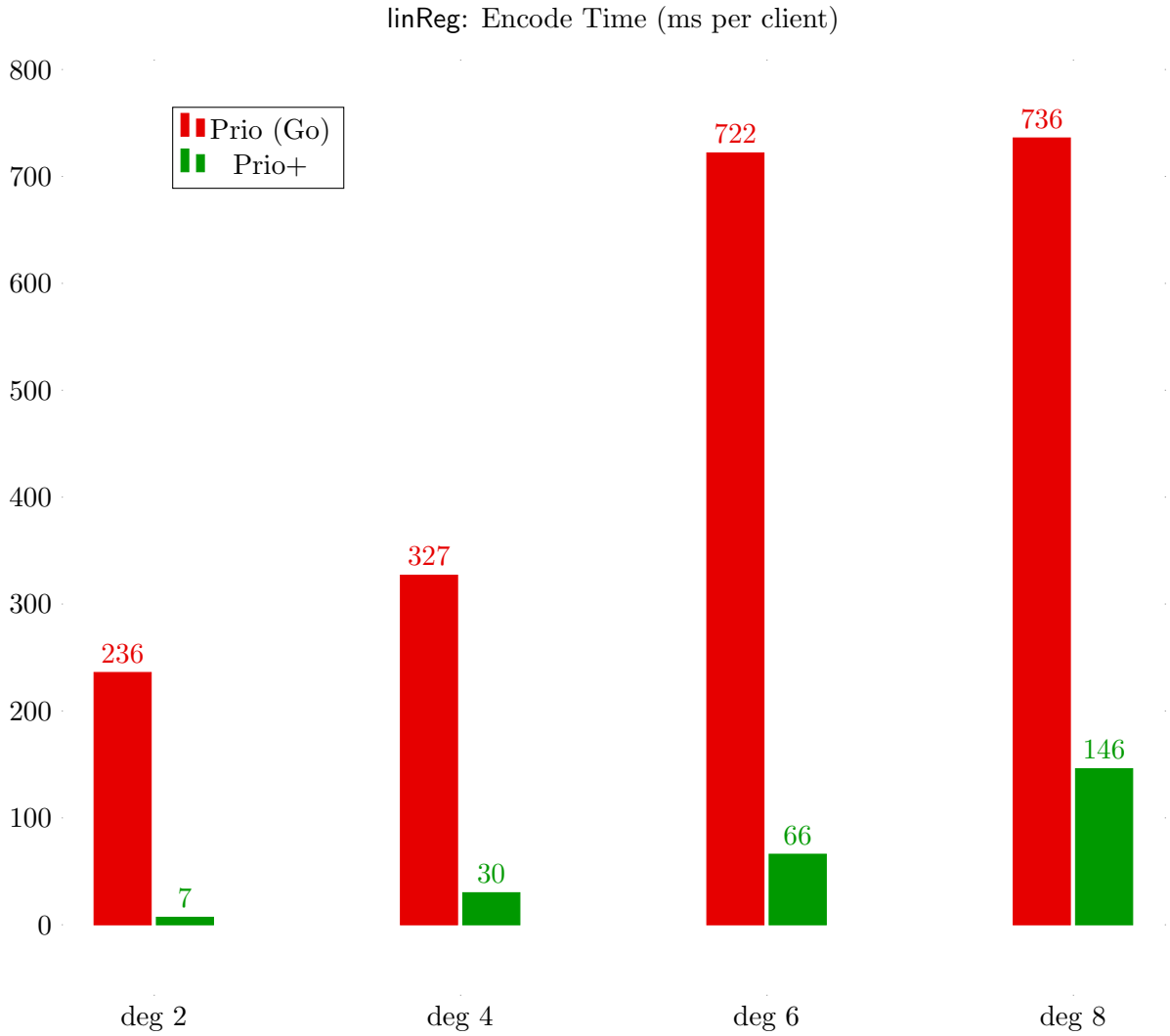

Figure 9: This chart shows the time necessary for a single client P_i holding private value $\vec{x}_i = (x_i^{(0)}, \ldots, x_i^{(d)})$ to encode that value, compute any necessary additional proofs, and secret-share both the encoding and the proof(s) when executing a protocol to compute $\mathsf{linReg}(\vec{x}_1, \ldots, \vec{x}_n)$. Each $x_i^{(k)}$ is an 8-bit integer. Results are in milliseconds and data is arranged according to the degree d of each $\vec{x}_i$.

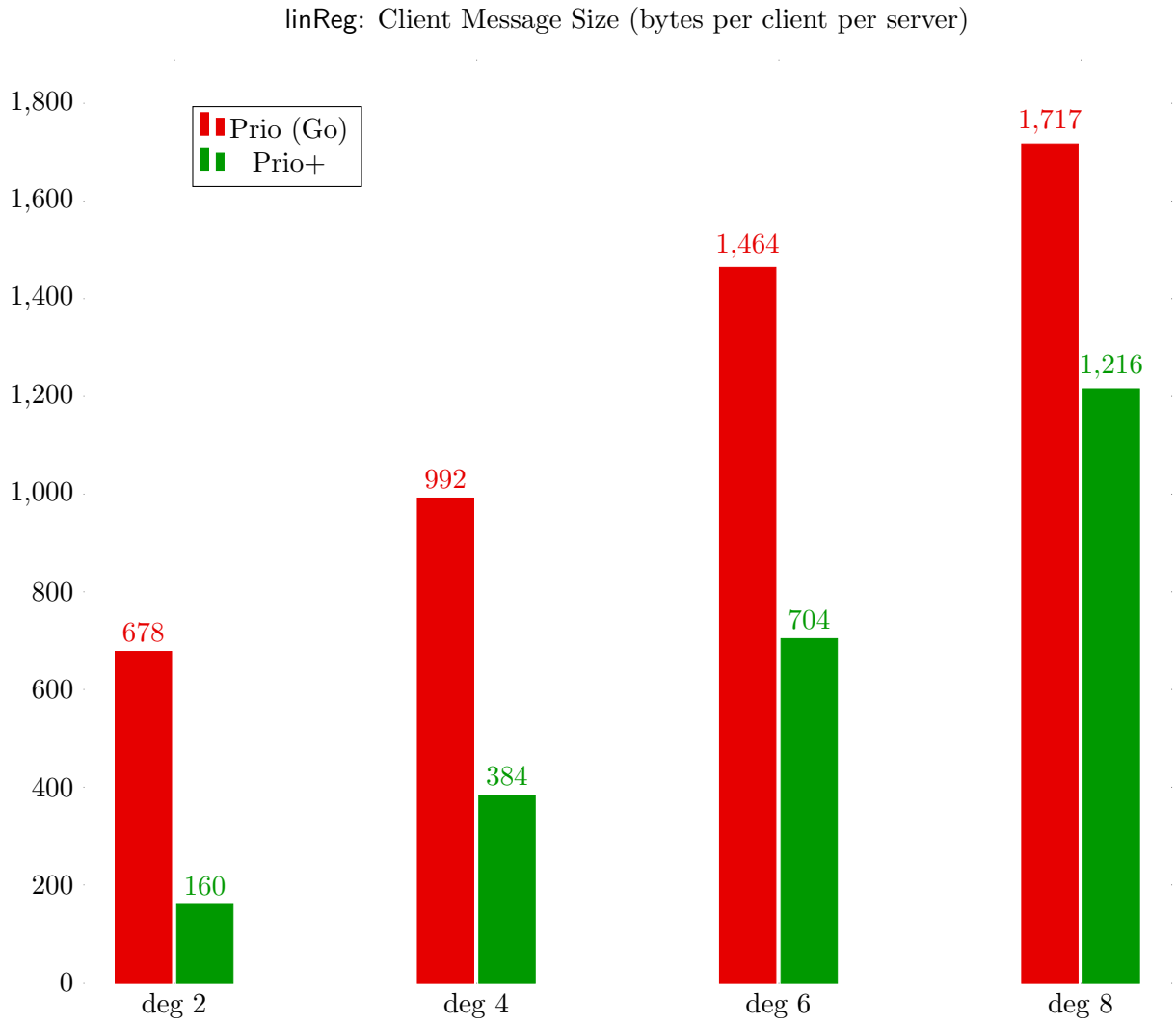

Figure 10: This chart shows the size of the message sent by client P_i (holding private value $\vec{x}_i = (x_i^{(0)}, \ldots, x_i^{(d)})$) to each server when executing a protocol to compute $\mathsf{linReg}(\vec{x}_1, \ldots, \vec{x}_n)$. Each $x_i^{(k)}$ is an 8-bit integer. Results are in milliseconds and data is arranged according to the degree d of each $\vec{x}_i$.

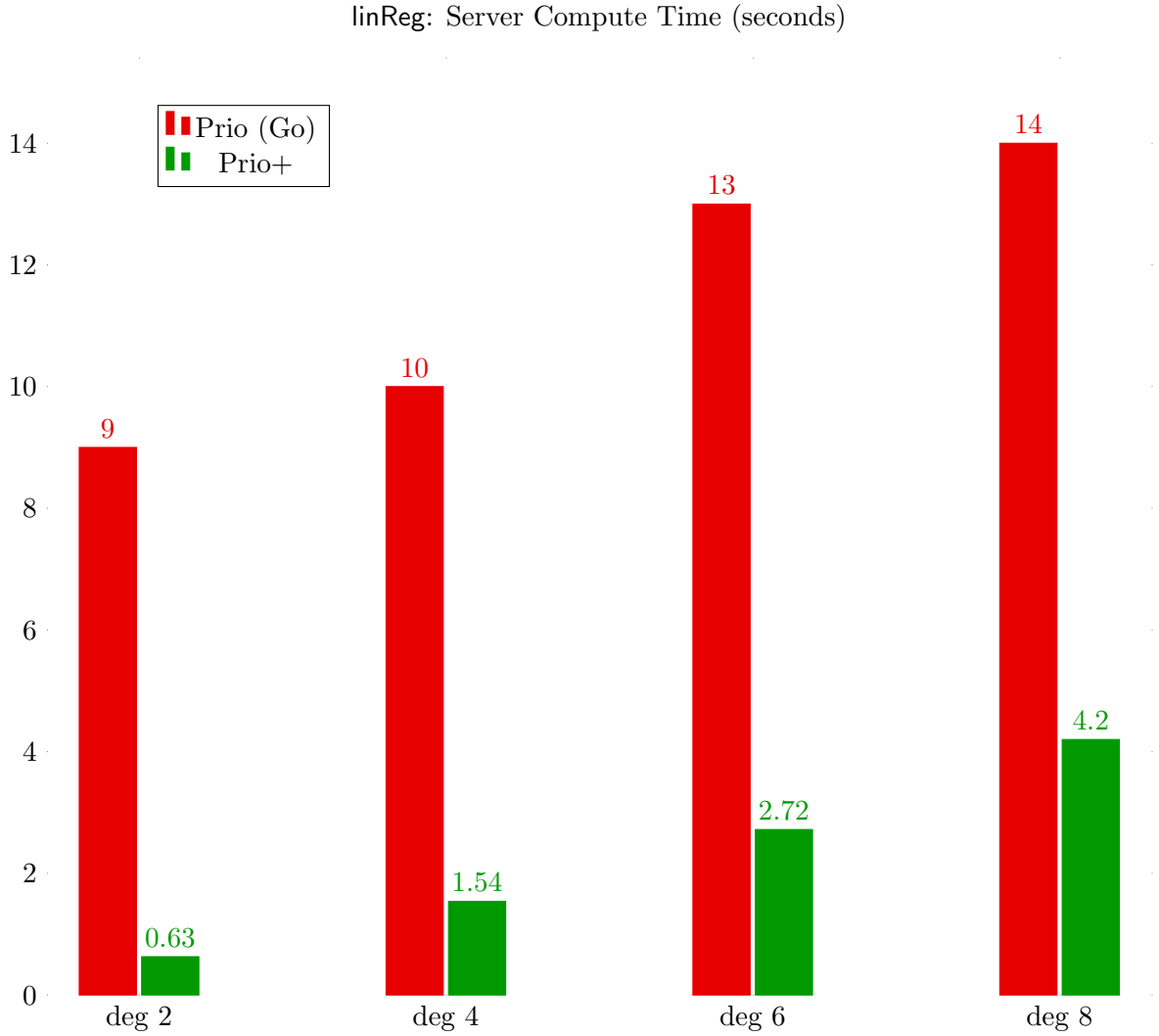

Figure 11: This chart shows the total server time to compute $\mathsf{linReg}(\vec{x}_1, \ldots, \vec{x}_n)$, where each of the 50,000 clients P_i holds private value $\vec{x}_i = (x_i^{(0)}, \ldots, x_i^{(d)})$ and each $x_i^{(k)}$ is an 8-bit integer. Results are in seconds and data is arranged according to the degree d of each $\vec{x}_i$.

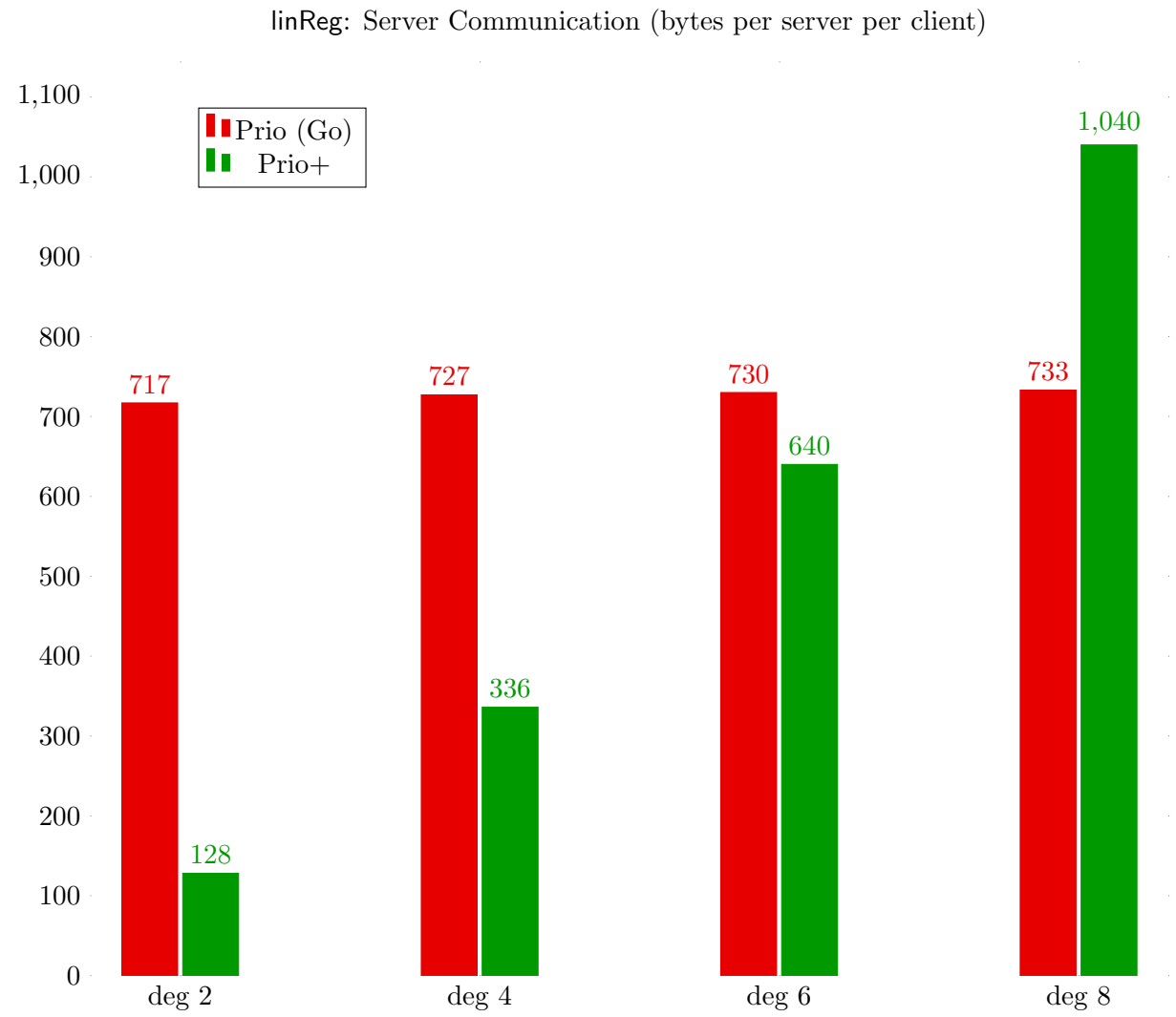

Figure 12: This chart shows the average bytes communicated by each server when executing a protocol to compute $\mathsf{linReg}(\vec{x}_1, \ldots, \vec{x}_n)$, where client P_i holds private value $\vec{x}_i = (x_i^{(0)}, \ldots, x_i^{(d)})$ and each $x_i^{(k)}$ is an 8-bit integer. Results are in bytes and data is arranged according to the degree d of each $\vec{x}_i$.

Analysis: Prio+ provides significant benefits to the client in computing linReg at a minimal server-side cost. Client encode time and message size are dramatically decreased, although the difference in client message size decreases as the degree increases. This is because as the degree increases, bit-length verification represents a smaller proportion of the multiplication gates in the VALID circuit. Once again we have an added benefit of decreased server computation, particularly for lower degree regression. Based on this trend, we expect the server computation in Prio+ to exceed that of Prio beyond degree 8 linear regression. We see a similar trend in terms of server communication, where Prio+ servers communicate less for lower degree computation, but that communication grows with the degree whereas Prio's server communication remains constant.

11.4 Data: Offline Pre-computation

In order to enable efficient share conversion, Prio+ requires some pre-computed data which is independent of client data. In particular, our novel edaBit-based share conversion protocol requires a b-bit edaBit for each b-bit integer to be converted. Thus, to evaluate b-bit SUM for 50,000 client submissions, we require 50,000 b-bit edaBits. The main cost in generating these edaBits lies in generating multiplication triples and OT correlations, which are used for combining the servers' private edaBits into public edaBits whose underlying value is unknown to both servers. The most efficient method for generating multiplication triples in the semi-honest model is based on homomorphic encryption. We use a semi-honest version of the methods described in [12] to generate multiplication triples. We implemented our protocol in PALISADE v1.10.4 (https://palisade-crypto.org/). We generated silent OT correlations using silent OT extension according to [6] using libOTe [30]. We measured how long the resulting edaBit generation procedure takes to generate 100,000 8-bit edaBits for with arithmetic shares in $\mathbf{Z}_p$ where p is a 48-bit prime, the largest prime which can be supported by PALISADE. The result was that our two large AWS servers were able to generate 100,000 8-bit edaBits in an average of 12.53 seconds.

We also measured how long Prio+ servers require to compute complex statistics if its pre-computed data cache is depleted and it must compute everything on the fly. This only affects two statistics (VAR and linReg) since they require edaBit-based share conversion into a field.

Prio+ can compute VAR for 10,000, 50,000, and 100,000 clients holding 8-bit inputs with no cache of pre-computed data in 0.78, 2.50, and 5.01 seconds respectively. We cannot compare these to Prio because it does not currently support VAR.

We compare Prio+ without pre-computation to Prio when computing linReg for 50,000 clients holding degree d vectors of 8-bit inputs for $d = 2, 4, 6, 8$. Our conclusion is that even without access to pre-computed data, Prio+ outperforms Prio for degree $d = 2$ computations, but for larger degrees $d = 4, 6, 8$ that savings is drowned out by the time to compute edaBits on the fly. This is understandable, as the larger d becomes, the more edaBits are necessary, and edaBits are the most expensive resource in Prio+. However, Prio+ is still on the same order of magnitude (at worst 4x slower) as Prio for all measured cases. Prio+ can thus be expected to perform comparably to Prio even in the rare situation where caches of pre-computed data are completely depleted. The full comparison data can be found below.

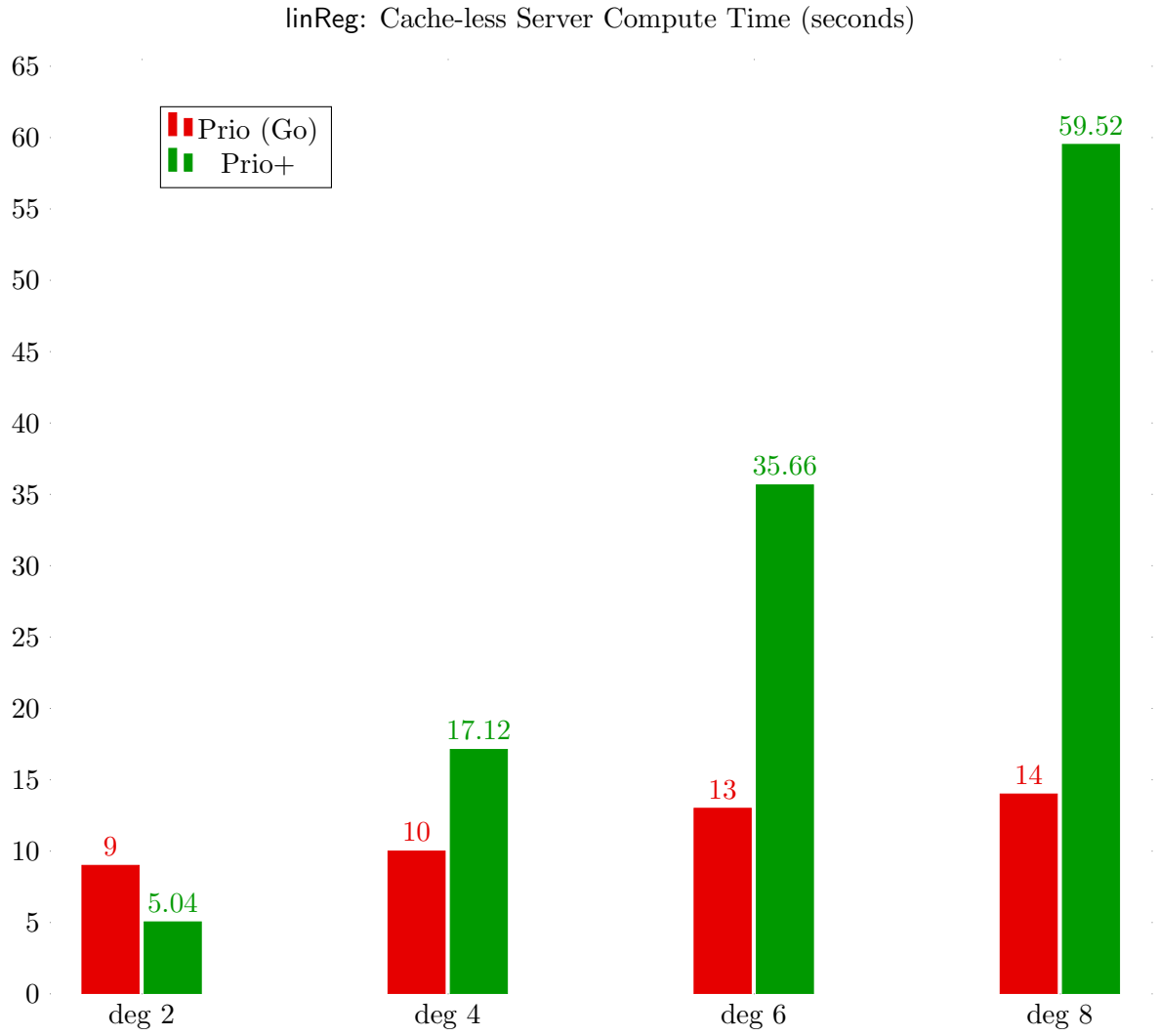

Figure 13: This chart shows the total time for both servers, without access to pre-computed data, to compute $\mathsf{linReg}(\vec{x}_1, \ldots, \vec{x}_n)$, $n = 50,000$, where each client P_i holds private value $\vec{x}_i = (x_i^{(0)}, \ldots, x_i^{(d)})$ and each $x_i^{(k)}$ is an 8-bit integer. Results are in seconds and data is arranged according to the degree d of each $\vec{x}_i$.

12 Conclusions and Future Work

By leveraging properties of Boolean secret-sharing, Prio+ effectively restructures the original Prio protocol to privately compute aggregate statistics with minimal burden on the client. Often the computational burden on the servers is also reduced, occasionally at the cost of increased server communication. The scale of these performance improvements depends heavily on the statistic being computed. For some statistics ($\mathsf{AND}, \mathsf{OR}, \mathsf{MAX}, \mathsf{MIN}$), Prio+ significantly reduces the burden on both clients and servers across the board. For others, particularly linear regression, the improvements in client performance and server computation come with a moderate increase in server bandwidth usage. These costs are only apparent, however, as the size of client inputs becomes large. For small and often practical client values, Prio+ still outperforms Prio across all examined metrics. Prio+ does require an additional offline pre-computation phase which enables efficient share conversion. However, this computation is relatively inexpensive and can be done during times when data is not being collected and servers are otherwise idle.

In the future, we wish to expand Prio+ to compute additional aggregate statistics. We would particularly like to identify statistics similar to $\mathsf{AND}, \mathsf{OR}, \mathsf{MAX}$, and MIN for which neither SNIPs nor share conversion are required. We are also interested in optimizing our offline pre-computation phase to generate edaBits more efficiently, as these are a new primitive which are relatively unexplored in the semi-honest setting and we believe they can be generated at a cheaper cost.

Acknowledgments

Supported in part by DARPA under Cooperative Agreement No: HR0011-20-2-0025, NSF Grant CNS-2001096, US-Israel BSF grant 2015782, Google Faculty Award, JP Morgan Faculty Award, IBM Faculty Research Award, Xerox Faculty Research Award, OKAWA Foundation Research Award, B. John Garrick Foundation Award, Teradata Research Award, Lockheed-Martin Corporation Research Award, NSF NRT HDR Grant NSF DGE-1829071, UCLA Connection Lab and Sunday Group, Inc. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, the Department of Defense, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright annotation therein. This paper was prepared in part for information purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

References

[1] Vpriv: Protecting privacy in location-based vehicular services. In *18th USENIX Security Symposium (USENIX Security 09)*, Montreal, Quebec, August 2009. USENIX Association.

[2] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining lsss and garbled circuits in scale. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'19, page 33–44, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[4] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *Annual Cryptology Conference*, pages 663–680. Springer, 2012.

[5] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. Cryptology ePrint Archive, Report 2019/188, 2019. `https://eprint.iacr.org/2019/188`.

[6] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. Cryptology ePrint Archive, Report 2019/1159, 2019. `https://eprint.iacr.org/2019/1159`.

[7] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 387–416, Cham, 2020. Springer International Publishing.

[8] 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. `https://crypto.stanford.edu/prio/paper.pdf`.

[9] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *Theory of Cryptography*, pages 342–362, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. `https://link.springer.com/content/pdf/10.1007%2F978-3-540-30576-7_19.pdf`.

[10] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, pages 13–25, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[11] 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://eprint.iacr.org/2012/642`.

[12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 643–662, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Santiago Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proceedings of the First ACM Workshop on Smart Energy Grid Security*, SEGS '13, page 75–80, New York, NY, USA, 2013. Association for Computing Machinery.

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

[15] Tariq Elahi, George Danezis, and Ian Goldberg. Privex: Private collection of traffic statistics for anonymous communication networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1068–1079, New York, NY, USA, 2014. Association for Computing Machinery.

[16] K. Emura, H. Kimura, T. Ohigashi, T. Suzuki, and L. Chen. Privacy-preserving aggregation of time-series data with public verifiability from simple assumptions and its implementations. *The Computer Journal*, 62(4):614–630, 2019.

[17] Úlfar Erlingsson, Aleksandra Korolova, and Vasyl Pihur. RAPPOR: randomized aggregatable privacy-preserving ordinal response. *CoRR*, abs/1407.6981, 2014.

[18] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. Springer-Verlag, 2020.

[19] Giulia C. Fanti, Vasyl Pihur, and Úlfar Erlingsson. Building a RAPPOR with the unknown: Privacy-preserving learning of associations and data dictionaries. *CoRR*, abs/1503.01214, 2015.

[20] James Glanz, Jeff Larson, and ANDREW W Lehren. Spy agencies tap data streaming from phone apps. *New York Times*, 2014.

[21] Oded Goldreich. *Foundations of Cryptography: Volume 1*. Cambridge University Press, USA, 2006.

[22] Andrew Hilts, Christopher Parsons, and Jeffrey Knockel. Every step you fake: A comparative analysis of fitness tracker privacy and security. *Open Effect Report*, 76(24):31–33, 2016.

[23] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 145–161, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[24] Tobias Jeske. Floating car data from smartphones: What google and waze know about you and how hackers can control traffic. *Proc. of the BlackHat Europe*, pages 1–12, 2013.

[25] Marc Joye and Benoît Libert. A scalable scheme for privacy-preserving aggregation of time-series data. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography and Data Security*, pages 111–125, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[26] Josh Keller, KR Lai, and Nicole Perlroth. How many times has your personal information been exposed to hackers. *New York Times (July 29, 2015)*, 2015.

[27] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches. *CoRR*, abs/1508.06110, 2015.

[28] Raluca Ada Popa, Andrew J. Blumberg, Hari Balakrishnan, and Frank H. Li. Privacy and accountability for location-based aggregate statistics. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 653–666, New York, NY, USA, 2011. Association for Computing Machinery.

[29] Deevashwer Rathee, Thomas Schneider, and K. K. Shukla. Improved multiplication triple generation over rings via rlwe-based ahe. Cryptology ePrint Archive, Report 2019/577, 2019. `https://eprint.iacr.org/2019/577`.

[30] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. `https://github.com/osu-crypto/libOTe`.

[31] Dragos Rotaru and Tim Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. Cryptology ePrint Archive, Report 2019/207, 2019. `https://eprint.iacr.org/2019/207`.

[32] Victor Shoup. Oaep reconsidered. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 239–259, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[33] Victor Shoup. A proposal for an iso standard for public key encryption. *IACR Cryptology ePrint Archive*, 2001:112, 01 2001.

[34] Ben Smith. Uber executive suggests digging up dirt on journalists. *BuzzFeed News*, 18, 2014.

[35] Gang Wang, Bolun Wang, Tianyi Wang, Ana Nika, Haitao Zheng, and Ben Y Zhao. Defending against sybil devices in crowdsourced mapping services. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 179–191, 2016.

A Full Protocol Descriptions

In this section we give detailed descriptions of the protocols which up until now have been described at a more informal level. We omit the protocol for MIN as it consists of a single call to MAX plus a single pre-processing step and a single post-processing step. We also omit the protocols for integer mean and standard deviation, as these too require a single post-processing step on top of the integer sum and variance protocols respectively.

Π_{sum}

Inputs: $x_i \in \mathbf{Z}_{2^{l'}}$ for $i \in [n]$.
Output: $\sum_i x_i$.

1. **Upload**:
   (a) Each client P_i computes $\mathsf{Share}_{\oplus,l'}(x_i) \longrightarrow [x_i]_L^B, [x_i]_R^B$ via Definition 4.3
   (b) Each P_i sends $[x_i]_L, [x_i]_R$ to S_L, S_R respectively.
2. **Verify Bit-Length**: Initially, $n' = n$. If a server receives a share which is not l bits in length from P_i (assume S_L w.l.o.g.):
   (a) S_L sends the index i to S_R.
   (b) Both servers discard $[x_i]^B$.
   (c) Both servers set $n' \leftarrow n' - 1$
3. **Convert Shares**: S_L and S_R jointly evaluate $\mathsf{B2A}_{l',2^l}(\{[x_i]_L^B, [x_i]_R^B\})$ on each of the n' valid pairs of Boolean shares as described in [14]. S_L receives as output $\{[x_i]_L^A\}_i$ and S_R receives as output $\{[x_i]_R^A\}_i$.
4. **Aggregate**: S_L locally adds all arithmetic shares into an accumulator A_L, initially zero. That is: $A_L \longleftarrow A_L + \sum_i [x_i]_L^A$. S_R analogously accumulates its arithmetic shares into $A_R \longleftarrow A_R + \sum_i [x_i]_R^A$.
5. **Publish**: Once all n' shares have been accumulated, S_L and S_R publish A_L and A_R to every client.
6. **Client Computation**: Clients output $A_L + A_R$.

Π_{and}

Inputs: $x_i \in \{0,1\}$ for $i \in [n]$.
Output: 1 if and only if $x_i = 1$ for all $i \in \{1, \ldots, n\}$.

1. **Upload**: Each P_i encodes their as input as:

   $\hat{x}_i = 0 \in \mathbf{Z}_{2^\lambda}$ if $x_i = 1$

   $\hat{x}_i = r \in \mathbf{Z}_{2^\lambda}$ if $x_i = 0$, where r is uniformly random

   P_i then computes $\mathsf{Share}_{\oplus,\lambda}(\hat{x}_i) = ([\hat{x}_i]^B_{L,\lambda}, \hat{x}_i]^B_{R,\lambda})$ as in Definition 4.3 and sends one share to each server.

2. **Verify Bit-Length**: Initially, $n' = n$. If some server, say S_L, receives from P_i $[x_i]^B_{L,\lambda}$ which is an m-bit integer, $m \neq \lambda$:

   (a) S_L sends the index i to S_R.

   (b) Both servers discard $[x_i]^B_\lambda$ (removing from accumulator if necessary).

   (c) Both servers set $n' \leftarrow n' - 1$

3. **Aggregate**: S_L and S_R hold accumulator values $A_L, A_R \in \mathbf{Z}_{2^\lambda}$, initially set to 0. Once a λ-bit share is sent to S_L by P_i, S_L *immediately* XORs it with A_L: $A_L \leftarrow A_L \oplus [\hat{x}_i]_L$. S_R does the same with its accumulator A_R upon receiving a valid share. If either server learns that a share already accumulated should be discarded, they simply XOR it with their accumulator again.

4. **Publish**: S_L publishes A_L to all clients, S_R publishes A_R to all clients.

5. **Client Computation**: Clients compute $A = A_L \oplus A_R \in \mathbf{Z}_{2^\lambda}$. If $A = 0$, clients output 1. Otherwise, they output 0.

Π_{or}

Inputs: $x_i \in \{0,1\}$ for $i \in [n]$.
Output: 0 if and only if $x_i = 0$ for all $i \in \{1, \ldots, n\}$.

1. **Upload**: Each P_i encodes their as input as:

   $\hat{x}_i = 0 \in \mathbf{Z}_{2^\lambda}$ if $x_i = 0$

   $\hat{x}_i = r \in \mathbf{Z}_{2^\lambda}$ if $x_i = 1$, where r is uniformly random.

   P_i then computes $\mathsf{Share}_{\oplus,\lambda}(\hat{x}_i) = ([\hat{x}_i]^B_{L,\lambda}, \hat{x}_i]^B_{R,\lambda})$ as in Definition 4.3 and sends one share to each server.

2. **Verify Bit-Length**: Initially, $n' = n$. If some server, say S_L, receives from P_i $[x_i]^B_{L,\lambda}$ which is an m-bit integer, $m \neq \lambda$:

   (a) S_L sends the index i to S_R.

   (b) Both servers discard $[x_i]^B_\lambda$ (removing from accumulator if necessary).

   (c) Both servers set $n' \leftarrow n' - 1$

3. **Aggregate**: S_L and S_R hold accumulator values $A_L, A_R \in \mathbf{Z}_{2^\lambda}$, initially set to 0. Once a λ-bit share is sent to S_L by P_i, S_L *immediately* XORs it with A_L: $A_L \leftarrow A_L \oplus [\hat{x}_i]_L$. S_R does the same with its accumulator A_R upon receiving a valid share. If either server learns that a share already accumulated should be discarded, they simply XOR it with their accumulator again.

4. **Publish**: S_L publishes A_L to all clients, S_R publishes A_R to all clients.

5. **Client Computation**: Clients compute $A = A_L \oplus A_R \in \mathbf{Z}_{2^\lambda}$. If $A = 0$, clients output 0. Otherwise, clients output 1.

Π_{max}

Inputs: $x_i \in \{0, \ldots, K-1\}$ for $i \in [n]$.
Output: $\mathsf{max}_i x_i$

1. **Upload**: Each P_i encodes their private x_i as $\hat{x}_i \in \mathbf{Z}_{2^K}^{\lambda}$, where:

   - $(\hat{x}_i)_j = r_j \in \mathbf{Z}_{2^\lambda}$ for $0 \leq j \leq x_i$, where r_j is a uniformly random λ-bit integer.
   - $(\hat{x}_i)_j = 0 \in \mathbf{Z}_{2^\lambda}$ for $x_i < j \leq K-1$

   We will use $(\hat{x}_i)_{j,k}$ to refer to the k'th component of $(\hat{x}_i)_{j,k})$. Each P_i then computes $\mathsf{Share}_{\oplus,\lambda K}(\hat{x}_i) = [\hat{x}_i]^B_{L,\lambda K}, [\hat{x}_i]^B_{R,\lambda K} \in \mathbf{Z}_{2^\lambda}^K$ as described in [14] and sends one share of $\hat{x}_i$ to each server.

2. **Verify Bit-Length**: Initially, $n' = n$. If some server, say S_L, receives from P_i $[x_i]^B_{L,\lambda K}$ which is an m-bit integer, $m \neq \lambda K$:

   (a) S_L sends the index i to S_R.
   (b) Both servers discard $[x_i]^B_{\lambda K}$ (removing from accumulator if necessary).
   (c) Both servers set $n' \leftarrow n' - 1$

3. **Aggregate**: S_L computes $A_L = \bigoplus_i [\hat{x}_i]^B_{L,\lambda K}$ and S_R computes $A_R = \bigoplus_i [\hat{x}_i]^B_{R,\lambda K}$.

4. **Publish**: S_L and S_R publish their accumulator values A_L and A_R to every client.

5. **Client Computation**: Clients compute

$$((s_1, \ldots, s_\lambda), \ldots, (s_{(K-1)\lambda+1}, \ldots, s_{K\lambda})) := A_L \oplus A_R$$

   Clients output the largest index $i \in \{1, \ldots, K\}$ such that the substring $(s_{(K-i)\lambda+1}, \ldots, s_{K-i+1}\lambda)$ is not identically zero.

Π_{var}

Inputs: $x_i \in \mathbf{Z}_{2^l}$ for $i \in [n]$.
Output: $\mathsf{VAR}(x_1, \ldots, x_n) = \mathbf{E}[X^2] - (\mathbf{E}(X))^2$.

1. **Upload**:
   (a) Each P_i encodes their input as $\hat{x}_i = (x_i, x_i^2) \in \mathbf{Z}_{2^{3l}}$, where x_i is written using l-bits and x_i^2 using $2l$ bits. We will call these two components $\hat{x}_i^{(1)}$ and $\hat{x}_i^{(2)}$ respectively.
   (b) Each P_i then computes $\mathsf{Share}_{\oplus,3l}(\hat{x}_i) = [\hat{x}_i]^B_{L,3l}, [\hat{x}_i]^B_{R,3l} \in \mathbf{Z}_{2^{3l}}$ via 4.3. We refer to the first l bits of these shares as $[\hat{x}_i]^{B,(1)}_l$ and the last $2l$ bits of these shares as $[\hat{x}_i]^{B,(2)}_{2l}$.
   (c) Each P_i computes the circuit $\mathsf{Valid}_{\mathsf{VAR}}(\hat{x}_i)$ and constructs polynomials f, g, h representing the values on the input and output wires of each multiplication gate.
   (d) P_i sends one share of $[\hat{x}_i]^B_{3l}$ to each server, as well as one share of $[f(0)]^B$, one share of $[g(0)]^B$, and one share of $[h]^B$ (that is, one share of each coefficient of h). P_i also computes and sends shares of a triple $([a]^B, [b]^B, [ab]^B)$ for random $a, b \in \mathbf{Z}_{2^{3l}}$.
2. **Verify Bit-Length**: Initially, $n' = n$. If some server, say S_L, receives from P_i $[\hat{x}_i]^B_{L,3l}$ which is an m-bit integer, $m \neq 3l$:
   (a) S_L sends the index i to S_R.
   (b) Both servers discard $[x_i]^B_{3l}$ (removing from accumulator if necessary).
   (c) Both servers set $n' \leftarrow n' - 1$
3. **Convert Shares**: S_L and S_R jointly evaluate $\mathsf{B2A}_{l,p}(\{[\hat{x}_i]^{B,(1)}_{L,l}, [\hat{x}_i]^{B,(2)}_{R,l}\})$ as well as $\mathsf{B2A}_{2l,p}(\{[\hat{x}_i]^{B,(1)}_{L,2l}, [\hat{x}_i]^{B,(2)}_{R,2l}\})$ on each of the n' valid sets of Boolean shares as described in Section 9. S_L receives as output $\{[\hat{x}_i]^{A,(1)}_L\}$ and $\{[\hat{x}_i]^{A,(2)}_L\}$ and S_R receives as output $\{[\hat{x}_i]^{A,(1)}_R\}$ and $\{[\hat{x}_i]^{A,(2)}_R\}$.
4. **Verify Encoding**: For each valid $\hat{x}_i$ received, servers verify it is truly of the form (x, x^2) as follows:
   (a) Servers compute values on all other wires of the $\mathsf{Valid}_{\mathsf{VAR}}(\hat{x}_i)$ circuit via affine operations on the values they have already (input wires, input/output wires of all multiplication gates). For input shares, servers use arithmetic shares from the output of share conversion.
   (b) Servers use polynomial interpolation on these shares to compute secret-shares of $[f]$ and $[g]$.
   (c) Servers choose a random $r \in \mathbf{Z}_{2^l}$ and each server locally computes $[f(r)]^B, [r \cdot g(r)]^B$, and $[r \cdot h(r)]^B$.
   (d) Servers use the Boolean multiplication triple $([a]^B, [b]^B, [ab]^B)$ to compute $[r \cdot f(r) \cdot g(r)]^B$. From this and $[r \cdot h(r)]^B$, servers use affine operations to compute $[r \cdot (f(r) \cdot g(r) - h(r))]^B$.

(e) Servers reconstruct the value $r \cdot (f(r) \cdot g(r) - h(r))$. If this value is non-zero, servers reject the input $\hat{x}_i$.

5. **Aggregate**: S_L computes $A_L^{(1)} = \sum_i [\hat{x}_i]_L^{A,(1)}$ and $A_L^{(2)} = \sum_i [\hat{x}_i]_L^{A,(2)}$, and S_R computes $A_R^{(1)} = \sum_i [\hat{x}_i]_R^{A,(1)}$ and $A_R^{(2)} = \sum_i [\hat{x}_i]_R^{A,(2)}$.

6. **Publish**: S_L and S_R publish their accumulator values $A_L^{(1)}$, $A_L^{(2)}$, $A_R^{(1)}$, $A_R^{(2)}$ to every client.

7. **Client Computation**: P_i outputs $\frac{1}{n}(A_L^{(2)} + A_R^{(2)}) - \frac{1}{n^2}(A_L^{(1)} + A_R^{(1)})^2$.

Π_{linReg}

Inputs: $(x_i, y_i) \in \mathbf{Z}_{2^l} \times \mathbf{Z}_{2^l}$ for $i \in [n]$.
Output: $\mathsf{linReg}((x_1, y_1), \ldots, (x_n, y_n)) = (c_0, c_1)$, where $\hat{y}(x) = c_0 + c_1 x$ is the unique line which minimizes the sum of squares loss $\sum_i (y_i - \hat{y}(x_i))^2$.

1. **Upload**:
   (a) Each P_i encodes their input as $\hat{x}_i = (x_i, x_i^2, y_i, x_i y_i) \in \mathbf{Z}_{2^{6l}}$, where x_i, y_i are each written using l bits and $x_i^2, x_i y_i$ are written using $2l$ bits. We will call these four components (from left to right) $\hat{x}_i^{(j)}$, where $j \in \{1, 2, 3, 4\}$.
   (b) Each P_i then computes $\mathsf{Share}_{\oplus,6l}(\hat{x}_i) = [\hat{x}_i]^B_{L,6l}, [\hat{x}_i]^B_{R,6l} \in \mathbf{Z}_{2^{6l}}$ via Definition 4.3. We refer to the j'th component of these shares as $[\hat{x}_i]^{B,(j)}$.
   (c) P_i sends one share of $[\hat{x}_i]^B_{6l}$ to each server.
2. **Verify Bit-Length**: Initially, $n' = n$. If some server, say S_L, receives from P_i $[\hat{x}_i]^B_{L,6l}$ which is an m-bit integer, $m \neq 6l$:
   (a) S_L sends the index i to S_R.
   (b) Both servers discard $[x_i]^B_{6l}$ (removing from accumulator if necessary).
   (c) Both servers set $n' \leftarrow n' - 1$
3. **Convert Shares**: S_L and S_R jointly evaluate $\mathsf{B2A}_{l,p}(\{[\hat{x}_i]^{B,(1)}_L, [\hat{x}_i]^{B,(1)}_R\})$ as described in Section 9 on each of the n' valid pairs of Boolean shares, which returns $\{[\hat{x}_i]^{A,(1)}_L\}$ to S_L and $\{[\hat{x}_i]^{A,(1)}_R\}$ to S_R. They similarly compute $\mathsf{B2A}_{2l,p}(\{[\hat{x}_i]^{B,(2)}_L, [\hat{x}_i]^{B,(2)}_R\})$, $\mathsf{B2A}_{l,p}(\{[\hat{x}_i]^{B,(3)}_L, [\hat{x}_i]^{B,(3)}_R\})$, and $\mathsf{B2A}_{2l,p}(\{[\hat{x}_i]^{B,(4)}_L, [\hat{x}_i]^{B,(4)}_R\})$. S_L receives as output $\{[\hat{x}_i]^A_L\}$ and S_R receives as output $\{[\hat{x}_i]^A_R\}$, returning $\{[\hat{x}_i]^{A,(j)}_L\}$ to S_L and $\{[\hat{x}_i]^{A,(j)}_R\}$ to S_R for $j \in \{2, 3, 4\}$.
4. **Verify Encoding**: For each valid $\hat{x}_i$ received, servers verify it is truly of the form $(x_i, x_i^2, y_i, x_i y_i)$ analogously to the VAR protocol.
5. **Aggregate**: For each $j \in \{1, 2, 3, 4\}$, S_L computes $A^{(j)}_L = \sum_i [\hat{x}_i]^{A,(j)}_L$ and S_R computes $A^{(j)}_R = \sum_i [\hat{x}_i]^{A,(j)}_R$.
6. **Publish**: S_L and S_R publish their accumulator values $A^{(j)}_L$, $A^{(j)}_R$ for $j \in \{1, 2, 3, 4\}$ to every client as well as the value n'.
7. **Client Computation**: P_i computes $A^{(j)} = A^{(j)}_L + A^{(j)}_R$ for $j \in \{1, 2, 3, 4\}$. P_i computes the output via Equation 1, using the values $A^{(1)} = \sum_i x_i$, $A^{(2)} = \sum_i x_i^2$, $A^{(3)} = \sum_i y_i$, $A^{(4)} = \sum_i x_i y_i$, and $n' = n$.

Π_{frq}

Inputs: $x_i \in \{0, \ldots, K-1\}$ for $i \in [n]$.

Output: $\mathsf{FRQ}(x_1, \ldots, x_n) = \vec{h} \in \mathbf{Z}^K$

1. **Upload**:
   (a) Each P_i encodes their input as $\hat{x}_i = (\delta_{x_i}) \in \mathbf{Z}_{2^K}$, the impulse vector at x_i.
   (b) P_i computes $\mathsf{Share}_{\oplus,K}(\hat{x}_i) = [\hat{x}_i]^B_{L,K}, [\hat{x}_i]^B_{R,K}$ via Definition 4.3.
   (c) P_i sends one share of $[\hat{x}_i]^B_K$ to each server.
2. **Verify Bit-Length**: Initially, $n' = n$. If some server, say S_L, receives from P_i $[\hat{x}_i]^B_{L,K}$ which is m-bits, $m \neq K$:
   (a) S_L sends the index i to S_R.
   (b) Both servers discard $[x_i]^B_K$ (removing from accumulator if necessary).
   (c) Both servers set $n' \leftarrow n' - 1$
3. **Convert Shares**: For each index $0 \leq j < K$ and for each of the n' valid pairs of K-bit Boolean shares, S_L and S_R jointly evaluate $\mathsf{B2A}_{1,2^l}(\{([x_i]^B_{L,K})_j, ([x_i]^B_{R,K})_j\})$ as described in [14]. S_L receives as output $\{[x_i]^*_L\}_i$ and S_R receives as output $\{[x_i]^*_R\}_i$, length-K vectors of Arithmetic shares satisfying $[x_i]^*_L + [x_i]^*_R = x_i \in \mathbf{Z}_{2^K}$.
4. **Verify Encoding**:
   (a) *Verify Odd Parity*:
      i. For each P_i, S_L computes the parity bit $\rho_i = \bigoplus_j ([\hat{x}_i]^B_{L,K})_j$ and sends ρ_i to S_R. S_R similarly computes its own parity bit $\omega_i = \bigoplus_j ([\hat{x}_i]^B_{R,K})_j$.
      ii. S_R computes $\sigma_i = \rho_i \oplus \omega_i$, the parity of $\hat{x}_i$.
      iii. If $\sigma_i = 1$, S_R does not respond and the verification passes. If $\sigma_i = 0$, S_R responds with the index i and both servers discard $[\hat{x}_i]^B_K$.
   (b) *Verify Total Impulse Count*:
      i. S_L computes $\sigma_L = \sum_i \sum_j ([\hat{x}_j]^*_L)_i \in \mathbf{Z}_{2^K}$. S_R similarly computes $\sigma_R = \sum_i \sum_j ([\hat{x}_j]^*_R)_i$.
      ii. S_L sends σ_L to S_R.
      iii. S_R computes $\sigma = \sigma_R + \sigma_L$
      iv. If $\sigma = n'$, where n' is the number of non-discarded inputs currently being considered, the verification passes and S_R does not respond.
      v. If $\sigma \neq n'$, S_R responds with '0' indicating failure. If $n' > 1$, the servers partition the set of n' remaining players in half lexicographically and repeat this check recursively in parallel on inputs from players $\{P_1, \ldots, P_{n'/2}\}$ and $\{P_{n'/2+1}, \ldots, P_{n'}\}$. If $n' = 1$, and this one remaining player is P_i, both servers discard their shares $[\hat{x}_i]$ once S_R responds with '0' and set $n' \leftarrow n' - 1$.

5. **Aggregate**: For all n' remaining clients, S_L computes $A_L = \sum_i [\hat{x}_i]_L^*$ and S_R computes $A_R = \sum_i [\hat{x}_i]_R^*$.
6. **Publish**: S_L and S_R publish their accumulator values A_L and A_R to every client.
7. **Client Computation**: Clients output $A = A_L + A_R \in \mathbf{Z}_{2^K}$.

B Security Definitions

We use identical definitions of f-privacy, anonymity, and robustness as used in [8]. We give an informal definition of f-privacy, for the sake of readability, which captures the proper security properties. We use the standard notions of negligible functions and computational indistinguishability (see [21]). We often leave the security parameter implicit.

Definition B.1 (f-privacy)**.** Suppose the final aggregate includes data from n different clients. We say that the protocol is f-private if, for:

- any malicious server S_i
- any number of malicious clients $m \leq n$

there exists an efficient simulator that, for every choice of the honest clients' inputs $(x_1, \ldots, x_{n-m})$, takes as input:

- the public parameters to the protocol run (all participants' public keys, the description of the aggregation function f, the cryptographic parameters, etc.),
- the indices of the adversarial clients and server,
- oracle access to the adversarial participants, and
- the value $f(x_1, \ldots, x_{n-m})$,

and outputs a simulation of the adversarial participants' view of the protocol run whose distribution is computationally indistinguishable from the distribution of the adversary's view of the real protocol run.

As discussed in [8], the adversary learns the value $f(x_1, \ldots, x_{n-m})$ exactly. If the number of honest players is not sufficiently large, this can leak significant information to the adversary about honest players' inputs. It is therefore up to the servers to ensure that sufficiently many honest players' inputs are included in the aggregate. Our system is subject to the same denial of service and intersection attacks as Prio for the same reasons they give. For a full description, see [8].

Definition B.2 (Anonymity)**.** We say that a data-collection scheme provides *anonymity* if it provides f-privacy (by Definition B.1) for $f = \mathsf{SORT}$, where SORT is the function that takes n inputs and outputs them in lexicographically increasing order.

A scheme achieving anonymity by this definition leaks the entire list of honest clients' inputs $(x_1, \ldots, x_{n-m})$ to the adversary. However, the adversary still learns nothing about which client submitted which value x_i. We also have the following Lemma from [8] which tells us that providing f-privacy for any symmetric function is necessary and sufficient for anonymity by this definition (where a symmetric function is one that is independent of the order of its inputs).

Lemma 2. *Let $\mathcal{D}$ be a data-collection scheme that provides f-privacy (by Definition B.1) for a symmetric function f. That is, $f(x_1, \ldots, x_n) = f(x_{\pi(1)}, \ldots, x_{\pi(n)}$ for all permutations π on n elements. Then $\mathcal{D}$ provides anonymity.*

Since all aggregate statistics we compute are symmetric functions, once we prove a particular data-collection scheme is f-private, we will have that it also provides anonymity by this result.

Before we can define robustness, recall that each client in the system holds a value $x_i \in \mathcal{D}$, where $\mathcal{D}$ is some set of valid data items (single bits, m-bit integers, impulse vectors). The definition of robustness tells us that when all servers are honest, a set of malicious clients cannot influence the output beyond their ability to choose *valid* inputs.

Definition B.3 (Robustness)**.** Fix a security parameter $\lambda > 0$. We say that a protocol in our scheme provides *robustness* if, when both servers execute the protocol faithfully, for every number m of malicious clients (with $0 \leq m \leq n$), and for every choice of honest clients' inputs $(x_1, \ldots, x_{n-m}) \in \mathcal{D}^{n-m}$, the servers, with all but negligible probability in λ, output a value in the set:

$$\Big\{ f(x_1, \ldots, x_n) | (x_{n-m+1}, \ldots, x_n) \in \mathcal{D}^m \Big\}.$$

C Proofs of Security

In this section we give proofs of privacy and robustness for each protocol described in Section 8. As discussed in the previous section, [8] shows that privacy of any symmetric functionality also implies anonymity. Since each statistic given here is symmetric (does not depend on order of inputs), we get anonymity as a free corollary of privacy in each case.

C.1 Privacy

Our protocols amend the Prio protocol in a few ways. These amendments, however, do not significantly affect the privacy proofs. In particular, our shift from Arithmetic to Boolean secret-sharing affects nothing about the hiding property of the scheme, so any singular shares are still indistinguishable from random. The introduction of the OT-based B2A share conversion reveals no additional information besides the output based on the results of [14]. The output, from the view of a single corrupted server, is once again a single share of a secret value but now in the arithmetic scheme, again indistinguishable from random. Since these additional interactions between players are all efficiently simulatable, our protocols remain secure. We do have to, however, prove that our novel edaBit-based share conversion protocol is also private in this regard.

Our protocols fall into three categories.

- No share conversion, no SNIPs ($\Pi_{\mathsf{and}}, \Pi_{\mathsf{or}}, \Pi_{\mathsf{max}}, \Pi_{\mathsf{min}}$)
- Share conversion, no SNIPs ($\Pi_{\mathsf{sum}}, \Pi_{\mathsf{frq}}$)
- Share conversion and SNIPs ($\Pi_{\mathsf{var}}, \Pi_{\mathsf{linReg}}$)

Intuitively, protocols in the first category have the easiest proofs of privacy. In these protocols, inputs are encoded in a manner such that any Boolean vector of the proper length is a valid encoding, meaning once client data is split into Boolean secret-shares, all validation work by servers can be done locally by simply verifying the length of the shares. Thus the view of an adversary controlling a single server and arbitrarily many clients contains only a single share of any honest player's input, which is indistinguishable from random according to the hiding property of our secret-sharing scheme. The only other message that the adversary sees is the honest server's aggregated value A_R. This value, however, can be computed deterministically from the output of the function and A_L, which are known to the adversary. Each proof in this category will follow this structure.

The second category is almost identical to the first, except that the protocol calls $\mathsf{B2A}$ on the set of clients' Boolean shares as a subroutine. The privacy of this protocol is proven in [14], and concludes that neither server, except with negligible probability, learns anything besides the corresponding set of arithmetic shares. This, in combination with the above argument, proves privacy of this set of protocols.

Protocols in the third category rely on the SNIP construction of [8] due to the fact that their encoding involves a multiplicative relationship. This is the ideal application of SNIPs and thus we utilize them in this situation. However, we still use $\mathsf{B2A}$ and our bit-length verification technique independently of the SNIP procedure. Thus, the only additional piece necessary to prove privacy for these protocols is the proof of zero-knowledge for SNIPs given in Appendix D.2 of [8]. This guarantees that a single semi-honest server (and any number of misbehaving clients) learn nothing from the SNIP verification of an honest player's input besides the fact that the multiplicative relationship(s) hold. This, in combination with the above arguments, completes the proof of privacy for this third category.

We now give formal proofs of privacy for each protocol.

Theorem 3. *The protocol* Π_{and} *is* AND*-private, where* $x_i \in \{0,1\}$ *and* $\mathsf{AND}(x_1, \ldots, x_n) = 1 \iff \forall i\ x_i = 1$.

Proof. Suppose, without loss of generality, that the adversary A corrupts S_L as well as m clients $P_{n-m+1}, \ldots, P_n$ where $m \leq n$. Suppose honest players $P_1, \ldots, P_{n-m}$ hold inputs $x_1, \ldots, x_{n-m} \in \{0,1\}$. We must construct an efficient simulator which takes the value $\mathsf{AND}(x_1, \ldots, x_n)$ and plays the part of the honest players in the protocol to construct a simulated view V^* which is computationally indistinguishable from the adversary's real view in the protocol $\mathsf{view}(A) = \mathsf{view}(S_L, P_{n-m+1}, \ldots, P_n)$. A's actual view (excluding adversarial inputs, which can be perfectly simulated given oracle access to the adversarial clients) is precisely:

$$\{[\hat{x}_1]_L^B, \ldots, [\hat{x}_{n-m}]_L^B, A_R\}$$

The hiding property of the Boolean and Arithmetic secret-sharing schemes guarantees that a single share reveals nothing about the underlying encoded secret $\hat{x}_i$ and is thus indistinguishable from a random ring element. Thus, we simulate the honest players' shares seen by S_L by sampling random elements from the proper ring, in this case $\mathbf{Z}_2^\lambda$.

To simulate the value A_R, we examine two possible cases. If the output is $\mathsf{AND}(\cdot) = 1$, then the simulator chooses the simulated value $\hat{A}_R$ to be equal to A_L. This is the only possible value of A_R which would result in the proper output, $A_L \oplus A_R = 0$. Otherwise, if $\mathsf{AND}(\cdot) = 0$ the simulator chooses A_R uniformly at random from $\mathbf{Z}_2^\lambda$, resulting in the proper output $A_L \oplus A_R = \vec{r}$. We know $\hat{A}_R$ and A_R come from the same distribution because they are both uniformly random subject to the constraint of producing the correct output. In conclusion, the simulated adversarial view is precisely $V^* = \{r_1, \ldots, r_{n-m}, \hat{A}_R\}$, where $r_i \in \mathbf{Z}_2^\lambda$ is chosen at random and $\hat{A}_R$ is defined as above. □

Corollary 4. *The protocol* Π_{or} *is* OR*-private, where* $x_i \in \{0, 1\}$ *and* $\mathsf{OR}(x_1, \ldots, x_n) = 0 \iff \forall i$ $x_i = 0$.

Proof. The proof is identical except for the simulation of A_R. In this case, if the output is 0, the simulator chooses $\hat{A}_R = A_L$, and if the output is 1 it samples $\hat{A}_R$ randomly from $\mathbf{Z}_2^\lambda$. □

Theorem 5. *The protocol* Π_{max} *is* MAX*-private, where* $x_i \in \{0, \ldots, k-1\}$ *and* $MAX(x_1, \ldots, x_n) = \max\{x_1, \ldots, x_n\}$.

Proof. The proof is nearly identical to the previous proof for AND, but we provide a detailed proof regardless.

Again we suppose, without loss of generality, that the adversary A corrupts S_L as well as m clients $P_{n-m+1}, \ldots, P_n$ where $m \leq n$. Suppose honest players $P_1, \ldots, P_{n-m}$ hold inputs $x_1, \ldots, x_{n-m} \in \{0, \ldots, k-1\}$. We must again construct an efficient simulator which takes the value $\mathsf{MAX}(x_1, \ldots, x_n)$ and plays the part of the honest players in the protocol to construct a simulated view V^* which is computationally indistinguishable from the adversary's real view in the protocol $\mathsf{view}(A) = \mathsf{view}(S_L, P_{n-m+1}, \ldots, P_n)$. A's actual view (excluding adversarial inputs, which can be perfectly simulated given oracle access to the adversarial clients) is once again:

$$\{[\hat{x}_1]_L^B, \ldots, [\hat{x}_{n-m}]_L^B, A_R\}$$

Once again, the hiding property of our secret-sharing schemes guarantees that a single share reveals nothing about the underlying encoded secret $\hat{x}_i$ and is thus indistinguishable from a random ring element, this time in the ring $\mathbf{Z}_2^{\lambda \times k}$. Thus, we simulate the honest players' shares seen by S_L by sampling random elements from $\mathbf{Z}_2^{\lambda \times k}$.

To simulate the value A_R, we again utilize the output of the protocol. Suppose without loss of generality that $\mathsf{MAX}(x_1, \ldots, x_n) = t$, where $t \in \{0, \ldots, M-1\}$. The adversary simply chooses $\hat{A}_R$ at random from $\mathbf{Z}_2^{\lambda \times k}$ subject to the following constraint: $(A_L \oplus A_R)_j = 0$ for $j < t\lambda$ and $(A_L \oplus A_R)_k = 1$ for some $k \in \{t\lambda, \ldots, (t+1)\lambda - 1\}$. We once again know $\hat{A}_R$ and A_R come

from the same distribution because they are both uniformly random subject to the constraint of producing the correct output. In conclusion, the simulated adversarial view is precisely $V^* = \{r_1, \ldots, r_{n-m}, \hat{A}_R\}$, where $r_i \in \mathbf{Z}_2^{\lambda \times k}$ is chosen at random and $\hat{A}_R$ is defined as above.

□

Corollary 6. *The protocol* Π_{min} *is* MIN*-private, where* $\mathsf{MIN}(x_1, \ldots, x_n) = \min\{x_1, \ldots, x_n\}$ *and* $x_i \in \{0, 1\}$.

Proof. The only interaction in Π_{min} is a single call to the Π_{max} protocol, whose privacy was established via the previous theorem. □

This concludes proofs of privacy for our first class of protocols which do not require share conversion. Although the next few proofs involve share conversion, they only require OT-based share conversion, so we simply rely on the proof of privacy from [14].

Theorem 7. *The protocol* Π_{sum} *is* SUM*-private, where* $x_i \in \mathbf{Z}_2^{l'}$ *and* $\mathsf{SUM}(x_1, \ldots, x_n) = \sum x_i$ *(as integers in* $\mathbf{Z}$*).*

Proof. Suppose, without loss of generality, that the adversary A corrupts S_L as well as m clients $P_{n-m+1}, \ldots, P_n$ where $m \leq n$. Suppose honest players $P_1, \ldots, P_{n-m}$ hold inputs $x_1, \ldots, x_{n-m} \in \mathbf{Z}_{2^{l'}}$. We must construct an efficient simulator which takes the value $\mathsf{SUM}(x_1, \ldots, x_n) = \sum_i x_i \in \mathbf{Z}$ and plays the part of the honest players in the protocol to construct a simulated view V^* which is computationally indistinguishable from the adversary's real view in the protocol $\mathsf{view}(A) = \mathsf{view}(S_L, P_{n-m+1}, \ldots, P_n)$. The underlying B2A protocol is $\mathcal{F}_{b2a}$-private (see [14]), so the adversary only learns the output of B2A and nothing else. Thus A's actual view (excluding dishonest players' inputs) is precisely:

$$\{[x_1]_L^B, \ldots, [x_{n-m}]_L^B, [x_1]_L^A, \ldots, [x_{n-m}]_L^A, A_R\}$$

Based on the hiding property of the Boolean and Arithmetic secret-sharing schemes, a single share in either scheme reveals nothing about the underlying secret and is thus indistinguishable from a random element of the corresponding ring. Thus, we simulate the honest players' shares seen by S_L by sampling random elements of the proper length, l' bits for the Boolean shares and l bits for the Arithmetic shares. $\hat{A}_R$ can be computed as $\mathsf{SUM}(x_1, \ldots, x_n) - A_L$, since $A_L + A_R = \mathsf{SUM}(\cdot)$. Thus the simulated view is $V^* = \{r_1, \ldots, r_n - m, r'_1, \ldots, r'_{n-m}, \hat{A}_R\}$, where $r_i \in \mathbf{Z}_{2^{l'}}$ and $r'_i \in \mathbf{Z}_{2^l}$ are chosen at random. □

Note that this proves privacy of our bit sum protocol as a corollary by simply substituting $l' = 1$.

Corollary 8. *The protocol* Π_{mean} *is* MEAN*-private, where* $x_i \in \mathbf{Z}_{2^{l'}}$ *and* $\mathsf{MEAN}(x_1, \ldots, x_n) = (\frac{1}{n}\sum_i x_i, n)$.

Proof. The Π_{mean} protocol is identical to the Π_{SUM} protocol except that servers also release to clients in the clear the number of players n' who submitted valid inputs so that they can properly compute the mean. The adversary's view and the simulated view are identical. □

Theorem 9. *The protocol* $\Pi_{\mathsf{frq}}(x_1, \ldots, x_n)$ *is* FRQ*-private, where* $x_i \in \{0, \ldots, k-1\}$ *and* $\mathsf{FRQ}(x_1, \ldots, x_n = \vec{h} - (f_1, \ldots, f_k) \in \mathbf{Z}_{n+1}^k$*, where* $f_j = |\{x_i : x_i = j\} \leq n$ *is the frequency of input* $j \in \mathbf{Z}_k$.

Proof. We once again assume A corrupts S_L as well as the last m clients. If we ignore the two additional verification steps ensuring that each input is an impulse vector, the adversary's view is once again

$$\mathsf{view}(A) = \{[x_1]_L^B, \ldots, [x_{n-m}]_L^B, [x_1]_L^A, \ldots, [x_{n-m}]_L^A, A_R\}$$

just as in the SUM protocol. Simulating the S_L shares is once again done via random sampling in the rings $\mathbf{Z}_{2^{l'}}$ and $\mathbf{Z}_{2^l}$ for the Boolean and Arithmetic shares respectively. Simulating A_R is once again done using the output of the protocol and A_L. Specifically, $\hat{A}_R = \vec{h} \oplus A_L$. Thus, to prove privacy, we need only prove the privacy of these two additional verification steps.

The first such step is a parity check on each encoded input $\hat{x}_i$ to make sure it contains an odd number of 1's. This is accomplished in the clear by having each server computing the parity of their share (call these parity bits $p_{i,L}$ and $p_{i,R}$) and taking the direct sum $p_i = p_{i,L} \oplus p_{i,R}$. Note that for any honest player P_i, we must have $p_i = 1$. Similarly, for any corrupted player P_j, our simulator knows the value p_j exactly. Thus, we can simulate all $p_{i,R}$ exactly by computing $p_{i,R} = p_i \oplus p_{i,L}$.

The second check is to ensure that no player submitted multiple impulses. To do this, servers compute $\mathsf{B2A}$ on each component of each secret-shared "impulse." Based on the privacy of $\mathsf{B2A}$ from [14], this does not reveal any information besides the output, the corresponding Arithmetic shares. The servers sum all vectors together and sum the components of the final vectors to get arithmetic shares of the total number of impulses submitted by all clients. Note that this value is is simply $n - c$, where c is the number of adversarial clients who submitted invalid inputs. Since our simulator knows the value c, this can be simulated exactly. If $c = 0$, we are done. Otherwise, the process repeats recursively on smaller subsets of players. Note that no matter how small these subsets become, the sum $n' - c'$ of total impulses in any subset will never reveal anything about an honest player's input, since it is publicly known that all honest players submitted a single impulse. This step will only possibly reveal adversarial players' inputs. Thus this second check is again private.

Since every message seen by the adversary in these additional verification steps is efficiently simulatable, we conclude that the protocol is private. □

Before we can prove the privacy of Π_{var}, Π_{stddev}, and Π_{linReg}, we must prove the privacy of the novel edaBit-based share conversion protocol on which these protocols rely.

Theorem 10. $\mathsf{B2A}_{l,p}$*, as described in Appendix D, is* $\mathcal{F}_{B2A}$*-private.*

Proof. We assume without loss of generality that the adversary A corrupts S_L and that S_L is the participant who reconstructs the value $x + r$. Since we know that the protocol Mult^B reveals nothing except for its output, we know any communication during the execution of Mult^B can be simulated. Since this is the only communication in Step 1 of the protocol, we need only simulate communication in Step 2, which solely consists of the values s_i, which are the bits representing the value $x + r$. Note that both x and r are unknown to either player and r is a uniformly random value in $\mathbf{Z}_{2^l}$. Thus, since x is also unknown, $x + r$ appears uniformly random in $\mathbf{Z}_{2^l}$. This means that the bits s_i appear uniformly random and our simulator can itself choose uniformly random bits for the simulated values. Since the view of the adversary is efficiently simulatable, we conclude that $\mathsf{B2A}_{l,p}$ is $\mathcal{F}_{B2A}$-private. □

We now have the tools to prove that Π_{var}, Π_{stddev}, and Π_{linReg} are private.

Theorem 11. *The protocol* Π_{var} *is* $\hat{\mathsf{VAR}}$*-private, where* $x_i \in \mathbf{Z}_{2^l}$ *and* $\hat{\mathsf{VAR}}(x_1, \ldots, x_n) = (\mathsf{VAR}\{x_1, \ldots, x_n\}, \mathbf{E}\{x_1, \ldots, x_n\})$.

Proof. Our proof is analogous to the previous examples except for three departures. First, our proof additionally relies on the privacy of the SNIP verification procedure. The full proof of security can be found in Appendix D of [8]. Their conclusion is that as long as one server is honest, an adversary controlling the remaining servers (and any coalition of clients) learns nothing from the SNIP verification procedure. This is analogous to how we rely on the security of the OT-based $\mathsf{B2A}$ protocol from [14] to conclude that the adversary's learns nothing besides the output of the protocol, except with negligible probability. This tells us that the view of our adversary remains the same before and after SNIP verification.

The second departure is that this protocol relies on our novel edaBit-based share conversion protocol. Since we have already proven that this protocol is private, we simply use that result to ensure that the adversary learns nothing but the output during share conversion. The adversary's view (not including adversarial inputs) is thus:

$$\{\{[f_i(0)]_L\}, \{[g_i(0)]_L\}, \{[h_i]\}, \{[\hat{x}_1]_L^B, \ldots, [\hat{x}_{n-m}]_L^B, [\hat{x}_1]_L^A, \ldots, [\hat{x}_{n-m}]_L^A, A_R\}$$

To simulate the polynomial shares, we call the simulator used in the SNIPs security proof in Appendix D of [8] as a subroutine. For the rest of the shares, we simulate via random sampling once again based on the hiding property of the secret sharing schemes.

The third departure is that the value A_R, when combined with A_L, reveals $\mathbf{E}[X]$ in addition to $\mathsf{VAR}[X]$, where X is a uniform random variable over values $x_1, \ldots, x_n$. This is why our functionality $\hat{VAR}$ additionally reveals $\mathbf{E}[X]$, meaning our simulator receives this as input. From this and the variance, the simulator can reconstruct $\mathbf{E}[X^2]$ using the identity $\mathsf{VAR}[X] = \mathbf{E}[X^2] - (\mathbf{E}[X])^2$. Then, since $A_L + A_R = \mathbf{E}[X^2] - (\mathbf{E}[X])^2$ with overwhelming probability, the simulator can simulate A_R by simply computing $\hat{A}_R = \mathbf{E}[X^2] - (\mathbf{E}[X])^2 - A_L$.

□

Corollary 12. *The protocol* Π_{stddev} *is* $\hat{\mathsf{STDDEV}}$*-private, where* $\hat{\mathsf{STDDEV}}(X) = (\sqrt{\mathsf{VAR}(X)}, \mathbf{E}[X])$.

Proof. The protocol is identical to Π_{var} with an additional local operation applied on the client side. Since there is no additional communication, the $\hat{\mathsf{VAR}}$-privacy of Π_{var} implies the corollary. □

Theorem 13. *The protocol* Π_{linReg} *is* $\hat{\mathsf{linReg}}$*-private, where* $\hat{\mathsf{linReg}}$ *additionally outputs* n'*, the number of players whose shares were counted in the aggregate.*

Proof. This proof is completely analogous to the previous proof for Π_{var}. The only difference between the protocols, besides the length of encoded shares, is that two multiplicative relationships are being verified within the encoded inputs. These are both done using SNIPs, however, and so with overwhelming probability neither server learns any additional information based on Appendix D in [8]. Simulation of A_R, once n', c_0, c_1, and A_L are known, can be done via simple matrix operations on the matrix equation given in Section 6. □

C.2 Robustness

In this section, we give proofs that each of our protocols are robust. That is, no client can affect the output of the protocol beyond misreporting their private value as some other valid input value. Our first and second categories of protocols (with the exception of Π_{frq}) will have the simplest robustness proofs. This is because all Boolean strings of the proper length are valid encodings, meaning the bit-length is the only property which servers must verify, and doing so is trivial when client data is shared via the Boolean scheme. In the protocols which utilize SNIPs, our robustness relies on the SNIPs' soundness property, proven in Appendix D of [8]. It guarantees that a malicious prover has only a negligible probability of successfully submitting an invalid proof to the verifiers. This, in combination with the trivial verification of bit-length, guarantees that each input lies in the correct range and they obey the proper multiplicative relationships. The most involved proof of robustness is for the Π_{frq} protocol, since we must give an argument from scratch of the correctness of our verification procedure.

Theorem 14. *The protocol* $\Pi_{\mathsf{and}}(x_1, \ldots, x_n)$ *is robust, where* $x_i \in \{0, 1\}$.

Proof. Note that any λ-bit Boolean vector is a valid encoded input. Any player who does not submit a λ-bit Boolean vector to each server is ignored by the protocol, which is equivalent to having an input of zero. Any player who does submit a λ-bit Boolean vector to each server, call them $v_L, v_R \in \mathbf{Z}_{2^\lambda}$, has submitted an encoding of a valid input, since $v_L \oplus v_R$ is a λ-bit Boolean vector. Thus any misbehaving client must either submit a valid encoded input or be treated as if they submitted input 0, which satisfies the definition of robustness. □

Theorem 15. *The protocol* $\Pi_{\mathsf{or}}(x_1, \ldots, x_n)$ *is robust, where* $x_i \in \{0, 1\}$.

Proof. See proof of previous theorem. □

Theorem 16. *The protocol* $\Pi_{\mathsf{max}}(x_1, \ldots, x_n)$ *is robust, where* $x_i \in \{0, \ldots, M-1\}$.

Proof. Same as previous proofs, except that valid encoded inputs are $M \times \lambda$-bit Boolean vectors. □

Theorem 17. *The protocol* $\Pi_{\mathsf{min}}(x_1, \ldots, x_n)$ *is robust, where* $x_i \in \{0, \ldots, M-1\}$.

Proof. Same as previous proof, valid encoded inputs are $M \times \lambda$-bit Boolean vectors. □

Theorem 18. *The protocol* $\Pi_{\mathsf{sum}}(x_1, \ldots, x_n)$ *is robust, where* $x_i \in \mathbf{Z}_{2^{l'}}$, $l' < l$.

Proof. Note that once again, any l'-bit Boolean vector is a valid encoded input. Thus, as long as each server received an l'-bit Boolean vector, the client submitted a valid input. For the client to submit anything besides a valid input, he must send to at least one server something besides an l'-bit Boolean vector, at which point the servers will detect it with certainty, discard that clients' shares, and continue as if that client submitted a zero. □

Corollary 19. *The protocol* $\Pi_{\mathsf{mean}}(x_1, \ldots, x_n)$ *is robust, where* $x_i \in \mathbf{Z}_{2^{l'}}$, $l' < l$.

Proof. The protocol consists of one call to Π_{sum} and local operations, so robustness of Π_{sum} implies robustness of Π_{sum}. □

This concludes our trivial robustness proofs. The following proofs rely on the soundness property of SNIPs, for which we rely on the results of [8].

Theorem 20. *The protocol* $\Pi_{\mathsf{var}}(x_1, \ldots, x_n)$ *is robust, where* $x_i \in \mathbf{Z}_{2^l}$.

Proof. Clients must submit shares of the proper length by the argument given in previous proofs. The only additional factor in this protocol is that the encoded input must be of the form (x, x^2). This is accomplished via the use of SNIPs, and our robustness property relies on the soundness of SNIPs proven in Appendix D of [8]. Their result guarantees that no client has more than $\frac{2\mu+1}{2^l}$ probability of submitting an input not of the form (x, x^2) for which the SNIP verification succeeds, where μ is the number of multiplication gates in the Valid circuit. Here, $\mu = \mathsf{poly}(l)$, so this probability is negligible. Since any input whose SNIP verification fails will be ignored (which is equivalent to submitting the value 0), this implies that no adversarial client can affect the output beyond misreporting their private value except with negligible probability. Thus, Π_{var} is robust. □

Corollary 21. *The protocol* $\Pi_{\mathsf{stddev}}(x_1, \ldots, x_n)$ *is robust, where* $x_i \in \mathbf{Z}_{2^l}$.

Proof. The protocol consists of one call to Π_{var} and local operations, so robustness of Π_{var} implies robustness of Π_{stddev}. □

Theorem 22. *The protocol* $\Pi_{\mathsf{linReg}}(x_1, \ldots, x_n)$ *is robust, where* $x_i \in \mathbf{Z}_{2^l}$.

Proof. In this case, we must verify that the encoded input is of the form (x, x^2, y, xy). The proof is identical to the proof of robustness for Π_{var} except that the Valid circuit verifies both multiplicative relationships at once. That is, according to the soundness of SNIPs proven in Appendix D of [8], no client has more than a negligible probability ($\frac{2\mu+1}{2^l}$) of submitting an improperly encoded input for which the SNIP verification algorithm succeeds. Thus, by the same argument as Π_{var}, Π_{linReg} is robust. □

Theorem 23. *The protocol* $\Pi_{\mathsf{frq}}(x_1, \ldots, x_n)$ *is robust, where* $x_i \in \{0, \ldots, k-1\}$.

Proof. If an adversarial client submits anything besides an k-bit Boolean vector to either server, their input will be discarded. Otherwise, the only ways to submit an invalid encoding are to submit a zero vector or to submit multiple impulses. If P_i submits a zero vector, servers will discover it has an even parity when they compute the parity in the clear. Thus, the input will be discarded. Since all zero vectors are detected with certainty in this step, assume from this point onwards that the servers do not hold shares of any zero vector and that there are n' remaining pairs of shares. If P_i submits a vector x such that $(x)_i = (x)_j = 1$, then the sum of all the components in the sum of all vectors will be greater than n', since there are no zero vectors. This means the players will be partitioned lexicographically and the check will repeat recursively. In the base case of this recursion, P_i will be the only member of his partitioned set, and this check will reveal that the sum of the components in his input is larger than 1, and his input will be discarded. This happens with certainty. Thus, no client can submit an invalid input without their input being ignored, satisfying the definition of robustness. □

This concludes our proofs of security. We have shown that each protocol described in this paper is private (with at most the same modest leakage as Prio), robust, and anonymous.