High Throughput Secure MPC Over Small Population in Hybrid Networks (Extended Abstract)

Ashish Choudhury[⋆1] and Aditya Hegde[1]

International Institute of Information Technology Bangalore, India.
`ashish.choudhury@iiitb.ac.in, aditya.shridhar@iiitb.org`

Abstract. We study secure multi-party computation (MPC) among small number of parties, in *partially synchronous* and *completely asynchronous* settings. Prior works have only considered the *synchronous* setting. The setting considered in this paper is that of $n = 4$ parties with $t = 1$ corruption. In this setting, we present the following results.

- A perfectly-secure protocol in the *partially synchronous* setting with 2 synchronous rounds. Our protocol simultaneously enjoys the properties of *optimal resilience* and *optimal number* of synchronous rounds and it partially answers one of the open problems of Patra and Ravi (IEEE Transactions on Information Theory, 2018).
- A cryptographically-secure protocol in the *partially synchronous* setting with 1 initial synchronous round. Our protocol has *optimal resilience* and *optimal number* of synchronous rounds. The previous such protocol (Beerliová, Hirt and Nielsen, PODC 2010) requires expensive public-key machinery and associated zero-knowledge (ZK) proofs and it was left as an open problem to reduce the cryptographic setups of their protocol. Our protocol makes inroads into solving this problem, where we deploy only standard symmetric-key gadgets and completely shun ZK proofs. Our protocol also improves upon the protocol of Beerliová et al, in terms of communication complexity.
- A cryptographically-secure protocol in the *asynchronous* setting, relying only on symmetric-key primitives. It improves upon the previous best protocols (Choudhury et al, ICDCN 2015 and Cohen, PKC 2016), which deploy expensive public-key tools and ZK protocols.

Keywords: MPC, Asynchronous Networks, High-throughput

1 Introduction

A central concept in cryptography introduced by Yao [58] is that of secure *multi-party computation* (MPC), which states that any distributed computation among

[⋆] This research is an outcome of the R&D work undertaken in the project under the Visvesvaraya PhD Scheme of Ministry of Electronics & Information Technology, Government of India, being implemented by Digital India Corporation (formerly Media Lab Asia).

mutually-distrusting participants in the *presence* of a centralised trusted entity can also be performed in the *absence* of any such trusted entity by running a protocol among the participants. A MPC protocol allows a set of n mutually-distrusting parties to perform a joint computation on their inputs while keeping it as private as possible. The MPC problem has been widely studied in various flavours [36,9,53] and several interesting results have been obtained regarding the theoretical possibility and feasibility of secure MPC (see [44] and its references).

The last decade has seen a surge in developing MPC protocols with a small number of parties for many practical tasks such as privacy-preserving machine learning [52,24,54,16,57,23,47,49], data analysis for financial applications [12], sugar beet auctions [13], etc. The advantages of MPC protocols for a small population are their simplicity and practical efficiency [2,1,34,25,14]. Also, most interesting use cases of MPC involve only a small number of parties. Several well-known MPC frameworks such as VIFF [32] and Sharemind [11] are targeted only for a small number of parties. MPC with a small number of parties also helps with MPC for a large population via server-aided computation, where the large population outsource their data and computation to a small number of servers.

The world of MPC over small population can be divided into two categories. The first group of protocols offer *high throughput* and require very low communication complexity [2,1,34,25,14,24,41]. Based on secret-sharing mechanisms, these protocols perform very simple computations over some algebraic structure (typically a ring or a field). The downside is that the number of rounds of interaction among the parties required in the protocol is proportional to the multiplicative depth of the circuit representing the underlying function to be computed. Consequently, the number of rounds is *not* a constant and hence these protocols are suited for low-latency networks, where small bandwidth is the primary concern. The second class of protocols, based on garbled circuits, require a *constant* number of rounds of interaction [48,22,18,17] and are suited for *high-latency* networks. The downside of these protocols is that they involve a huge amount of computation and parties need to perform a lot of symmetric-key cryptographic operations. In this work, we focus on *high-throughput* protocols.

Synchronous vs Asynchronous Networks. All prior works for MPC over small population are in the *synchronous* setting, where the parties are synchronized by a *global* clock and there exists a *publicly known* upper bound on the message delays over the communication channels among the parties. That is, each party knows beforehand how long it has to wait for an expected message, and if the message does not arrive within that time-bound, then it can conclude that the sender is *corrupt*. Unfortunately, it is impossible to ensure such strict time-outs in real-world networks like the Internet, where the communication channels may have arbitrary delays. Such networks are more appropriately captured by the *asynchronous* communication setting [8,20], where there does not exist any global clock and the messages of the parties can be *arbitrarily* delayed. The only guarantee in this model is that any message sent by a party is *eventually* delivered and is not "stuck" in the network forever. Apart from a better modelling of real-world networks, asynchronous protocols have the advantage of running at the *actual* speed of the underlying network. More specifically, for a synchronous protocol, the participants have to pessimistically set the global delay Δ to a large value to ensure that the messages sent by every honest party at the beginning of each round reaches their destination within the Δ time frame. But if the *actual* delay δ is such that $\delta << \Delta$, then the protocol fails to take advantage of the faster network and its running time will be proportional to Δ.

The biggest challenge in designing a *completely* asynchronous protocol is that a slow sender *cannot* be distinguished from a corrupt sender, i.e., if a party does not receive an expected message, then unlike the synchronous setting, it cannot decide whether the sender is corrupt (and has not sent the message) or slow (a sent message has been delayed). Consequently, if t is the threshold on the number of corruptions in the system, then in an asynchronous protocol, a party can afford to receive messages from at most $n - t$ parties and has to proceed to the next step, to avoid endless waiting. However, in this process, the communication from up to t potentially honest parties may have to be ignored. Due to this phenomena, synchronous protocols fail completely when executed in a completely asynchronous environment, as the security of synchronous protocols depend upon the fact that the messages of *all* the honest parties are considered.

The theoretical possibility and feasibility of *asynchronous* MPC (AMPC) protocols have been studied in the past [8,10,5,50,28]. However, compared to their synchronous counterparts, AMPC protocols perform poorly in terms of resilience (namely the number of faults t which can be tolerated) and communication complexity (the total number of bits communicated by the honest parties). A midway approach to improve resilience and communication complexity is to design *hybrid* MPC (HMPC) protocols in a hybrid network setting [5,7,51] which is a "combination" of synchronous and asynchronous networks. Hybrid networks are assumed to be synchronous in the "beginning", for the first few rounds, followed by a completely asynchronous setting. Even though generic AMPC and HMPC protocols have been designed in the past, to the best of our knowledge the practical efficiency of AMPC and HMPC, especially over small population has not been studied. The main aim of this work is to initiate the study of AMPC and HMPC protocols over *small population* and with *high throughout.*

1.1 Our Results and Existing Works

Round Optimal Perfectly-secure HMPC. From [51], *perfectly secure* HMPC protocol tolerating a *computationally unbounded Byzantine* adversary corrupting up to t parties maliciously exists iff $t < \frac{n}{3}$ and any such protocol needs *at least* 2 synchronous rounds. However, there *does not* exist any perfectly-secure HMPC protocol with 2 synchronous rounds *and* with resilience $t < \frac{n}{3}$ and it was left as an open problem in [51] to design such a protocol. The previous best perfectly-secure HMPC protocol with $t < \frac{n}{3}$ requires 3 synchronous rounds [51]. In this work, we partially solve the open problem of [51], by presenting a perfectly-secure HMPC protocol for $n = 4$ and $t = 1$ with 2 synchronous rounds. Our protocol is both *round optimal* in terms of the number of synchronous rounds, as well as has *optimal resilience.* Apart from having the optimal number of synchronous rounds, our HMPC protocol significantly improves upon the communication cost *per multiplication gate* of the HMPC protocol of [51]. While the pre-processing phase of [51] needs a communication of 21804 and broadcast of 18900 field elements to generate a single shared multiplication triple, we are able to generate the same with a communication of 2733 and broadcast of 3024 field elements (we defer the analysis of [51] to the full version of the paper).

Cryptographic HMPC with Symmetric Key Primitives. HMPC with cryptographic-security tolerating a *computationally-bounded* Byzantine adversary is possible with 1 synchronous round and $t < \frac{n}{2}$ [7]. The protocol in [7] utilizes computationally expensive asymmetric-key machinery, such as threshold signatures, threshold homomorphic encryption and associated zero-knowledge (ZK) proofs. It was left as an open problem in [7] to reduce the cryptographic setup of their HMPC protocol. We make inroads in solving this open problem by presenting an HMPC protocol with one synchronous round for $t = 1$ and $n = 4$, where we rely *only* on symmetric-key primitives. More specifically, we use *only* pseudo-random functions (PRF) to generate shared randomness among the parties. Apart from completely shunning relatively expensive public-key machinery, we also completely shun ZK proofs in our protocol.

In addition to using simpler primitives, our protocol outperforms [7], in terms of communication complexity per multiplication gate. In [7], the synchronous round is used *only* for privately receiving the circuit-inputs of the parties and not for pre-computing "raw data" to aid in later evaluation of multiplication gates. Consequently, the evaluation of multiplication gates during the asynchronous phase involves a lot of interaction and computationally heavy ZK protocol instances. More specifically, the public-key machinery of [7] operates over a ring $\mathbb{Z}_N$ for some κ-bit modulus N, where κ is the computational security parameter. The protocol communicates *approximately* 723 elements from $\mathbb{Z}_N$ *plus* 156 instances of ZK proofs of various types, per multiplication gate (we defer the analysis of [7] to the full version of the paper). On the contrary, we leverage the synchronous phase to get the circuit-inputs of the parties, as well as to initiate the generation of shared and random multiplication triples [3], during a pre-processing phase. Later, the evaluation of a multiplication gate just needs two public reconstructions of secret-shared values. The pre-processing phase needs a communication of only 64 field elements over the point-to-point channels and a broadcast of 18 field elements, while the actual evaluation of a multiplication gate needs an *amortized* communication of 16 field elements.

Cryptographic AMPC with Symmetric Key Primitives. AMPC with cryptographic security and tolerating t Byzantine corruptions is possible iff $t < \frac{n}{3}$ [38,39,27]. Previous high-throughput AMPC protocols are based on public-key machinery and ZK proofs. We present an AMPC protocol for $n = 4$ and $t = 1$, based only on PRFs and shun all kinds of ZK proofs. The pre-processing phase requires an amortized communication of 64 field elements over the point-to-point channels and a broadcast of 18 field elements through asynchronous reliable broadcast, while the evaluation phase needs an amortized communication of 16 field elements per multiplication gate. This is to be compared with the communication-efficient AMPC protocol of [27] with $n = 4$ and $t = 1$, based on *threshold somewhat homomorphic encryption* (TSHE) [35,33,26] (which is computationally more expensive than threshold homomorphic encryption and PRFs) and associated ZK protocols. The amortized communication of [27] per multiplication gate is as follows: the pre-processing phase needs a broadcast of 4 elements from $\mathbb{Z}_N$ through asynchronous reliable broadcast and 12 instances of ZK proofs, for proving the knowledge of underlying plaintexts. The computation phase needs a communication of 24 elements from $\mathbb{Z}_N$ over the point-to-point channels and 24 instances of ZK proofs, for proving correctness of partial decryption of ciphertexts (the full analysis of [27] is deferred to the full version of the paper). In [29], a constant round AMPC protocol is presented assuming the existence of a threshold FHE (TFHE) scheme and related ZK proofs, where the communication complexity is independent of the number of multiplication gates in the circuit. Since the existence of TFHE is a *stronger* assumption than TSHE and PRFs, we do not compare our protocol with [29].

1.2 Techniques Used

Like all the previous works on high-throughput MPC, the main tool used in our protocols is secret-sharing (SS) where our protocols evaluate the underlying circuit (representing the function to be computed) ensuring that the value over each wire in the circuit is secret-shared. The protocols are divided into three phases, pre-processing/triple-generation phase, input phase and circuit-evaluation phase. During the pre-processing phase, the parties generate secret-shared random *multiplication triples* (also called as Beaver's triplets) [3], independent of the input to multiplication gates, which are later used in the circuit-evaluation phase to evaluate multiplication gates efficiently; at the cost of two public reconstructions per multiplication gate. The input phase is used to get the shared inputs of the parties for the circuit. For HMPC protocols, the pre-processing phase is executed in a hybrid environment, utilizing the synchronous round(s) available in the beginning, followed by completely asynchronous steps, while the circuit-evaluation phase is completely asynchronous. To ensure *input provision* (namely to get the inputs of all the parties for the computation, which is otherwise impossible in a completely asynchronous setting) for the HMPC protocols, the input phase is executed in parallel with the triple-generation phase, to utilize the synchronous rounds available at the beginning. For the AMPC protocol, all the three phases are completely asynchronous. While the approach of shared circuit-evaluation is not new, the challenges are to generate the triples efficiently, specially in the hybrid setting, deploying the *optimal* number of synchronous rounds.

The generic SS based MPC protocols typically employ Shamir's SS [55]. However, we utilize *replicated* SS (RSS), which has been used in recent works on high-throughput secure *three-party computation* (3PC) with one corruption [2]. We extend the notion of RSS to the case of 4 parties with one corruption, where the set of parties $\mathcal{P} = \{P_1, \ldots, P_4\}$ is divided into groups $\mathcal{G}_1, \ldots, \mathcal{G}_4$, with each $\mathcal{G}_i$ consisting of 3 parties $\mathcal{P} \setminus \{P_i\}$. A RSS of a value s then consists of shares $s_1, \ldots, s_4$, where $s = s_1 + \ldots + s_4$ and all the (honest) parties in $\mathcal{G}_i$ have the share s_i. There are several advantages of using RSS, compared to Shamir's SS. In the *perfectly-secure* setting, *verifiably* generating a RSS of a value s is relatively simpler, compared to that of verifiably generating a Shamir's SS of s. This is because the latter requires generating what is called as "two-level" SS of s, where each share of s has to be further Shamir-shared and then the parties use the "second-level" share-shares to verify if the Shamir-shares of s lie on a consistent polynomial. Generating the two-level SS of s calls for other primitives and consequently the overall process is highly expensive.

The process of generating a RSS is further simplified in the cryptographic setting if the parties have access to a trusted key set-up for PRFs. However, no such simplification is known for Shamir SS in the cryptographic setting, even with a key set-up for PRFs. Furthermore, reconstruction for RSS is more efficient compared to Shamir's SS. If P_i is a designated party to reconstruct a shared value s, then for Shamir-sharing, all the remaining three parties need to send their share to P_i. For RSS, two parties in $\mathcal{G}_i$ can send s_i to P_i, while the third party can send a hash of s_i (note that P_i will have the remaining shares of s, as it will be present in three groups). If there are many values to be reconstructed by P_i (which is the case while evaluating the multiplication gates), then one of the parties just needs to send a *single* hash value on behalf of all its shares and *only two* parties in $\mathcal{G}_i$ need to send their shares to P_i. So in the *amortized* sense, the communication required to let P_i reconstruct ℓ secret-shared values is 2ℓ field elements, compared to 3ℓ field elements required for Shamir SS.

We follow the efficient framework of [28] to generate shared Beaver's triplets during the pre-processing phase. The framework consists of two building blocks: a triple-sharing protocol and a triple-extraction protocol. The triple-sharing protocol allows a designated *dealer* to *verifiably* share multiplication triples, where the triples are random and private for an *honest* dealer. The triple-extraction protocol is used to "securely extract" a sharing of truly random and private multiplication triples from a collection of shared multiplication triples "contributed" by individual parties. While the triple-extraction protocol can be executed in a completely asynchronous environment we utilize the available synchronous round(s) for the triple-sharing protocol in HMPC.

The HMPC protocol of [51] also utilizes the framework of [28] by presenting a *robust* triple-sharing protocol with 3 synchronous rounds, which *always* generates a Shamir SS of multiplication triples for the dealer. On contrary, we utilize the party-elimination framework of [37,6] to do triple-sharing in 2 synchronous rounds. Namely, our protocol is *not* robust and may fail to generate any shared triple if the adversary strikes, but in that case, the parties identify a *public* "conflict set" of at most 3 parties, *always* containing the corrupt party. Consequently, the remaining party(ies) are *honest* and any one of them can be designated as a *trusted third party* (TTP) to receive the inputs of the parties in clear and compute the function output. We stress that this is required *only if* the triple-sharing protocol fails and outputs a conflicting set; else the triple-generation protocol outputs shared triples and parties proceed to shared circuit-evaluation.

We further stress that if the parties send their clear inputs to the TTP for circuit evaluation, then it cannot be considered as a breach of privacy, as the TTP is guaranteed to be an *honest* party, while the privacy requirement for an MPC protocol is with respect to the corrupt parties. A similar approach of identifying disputes among sufficiently many parties and using the left-over party(ies) as a TTP has been explored in some of the recent works in the domain of MPC for small population, but in the *synchronous* setting [18,16].

1.3 Experimental Results

We implement and benchmark our cryptographically secure AMPC protocol and evaluate the performance of the triple-generation phase. We benchmark our protocol by using it for secure prediction on a linear regression model and verify that asynchronous protocols run at the actual speed of the underlying network.

1.4 Other Related Works

We provide the highest level of security, namely that of *guaranteed output delivery* (GOD) [30], where the honest parties *always* receive the correct output irrespective of the behaviour of the corrupt parties. In the *synchronous* setting, weaker security notions such as *security with abort* have been considered for the sake of getting more efficient MPC protocols [1,34,24]. In these protocols, the honest parties may pre-maturely abort the protocol execution, without receiving the function output, if the corrupt parties deviate from honest behaviour. Typically this is done by comparing the protocol messages received from *all* the parties and aborting in case of any inconsistencies. It is not clear how these protocols can be extended to the asynchronous setting with termination guarantees since the messages of *all* the parties need not be available. Indeed, [45] proposes a generic perfectly-secure AMPC protocol with a robust circuit-evaluation phase, but with an *optimistic* pre-processing phase, where the parties *need not* terminate, even if a single corrupt party crashes.

In [32] a perfectly-secure MPC protocol with $t < n/3$ is presented in a *partially synchronous* network, where there exists a *synchronization point* and the network is assumed to be asynchronous before and after this point. The protocol *optimistically* and asynchronously runs any perfectly-secure *synchronous* pre-processing protocol with $t < n/3$ till the synchronization point and then the parties verify if *all* the "expected" messages (including the ones from the corrupt parties) have been delivered before the deadline set by the synchronization point. The synchronization point is set to ensure that the expected messages of honest parties are delivered within the deadline. However, the protocol does not achieve GOD since a corrupt party can always delay its messages arbitrarily causing the overall protocol to *abort* pre-maturely. Our hybrid model is completely different from the model of [32], where synchronization is assumed at the *beginning*, instead of the *middle* of a protocol. Moreover, our protocols ensure GOD.

In [31], a cryptographically-secure generic AMPC protocol is presented tolerating $t < n/3$ corruptions, completely based on symmetric-key primitives (namely PRFs), without relying on ZK protocols. However, [31] extend the notion of distributed garbling [4] to the asynchronous setting to achieve a constant "round" protocol. The resultant protocol is expensive in terms of computation and communication complexity. Since our goal is to achieve high throughput, we do not compare our protocols with [4]. The work of [40] gives information-theoretically secure MPC protocols with various security guarantees in the *synchronous* setting, tolerating $t = 1$ corruption. Their protocols are statistically-secure and not comparable with our perfectly-secure HMPC protocols.

1.5 Future Directions

Our perfectly-secure HMPC works *only* when $t = 1$. We leave it as a challenging open problem to generalize it for *any* $t < \frac{n}{3}$. Our cryptographically-secure HMPC and AMPC protocols *can* be generalized for any $t < \frac{n}{3}$, but the resultant protocols will require exponential (in n) communication and computation complexity due to the cost of generalizing the notion of RSS. A similar shortcoming prevails in the existing RSS-based MPC protocols in the synchronous setting [2,1,34,47,14] as well, which are tailor-made only to handle $t = 1$ corruption. Designing efficient high-throughput protocols (both in the hybrid as well as asynchronous setting) with cryptographic-security for any $t < \frac{n}{3}$ and that too with only symmetric-key primitives is left as a future work.

Due to space constraints, we defer the formal proofs of our protocols to the full version of the paper.

2 Preliminaries

We assume a set of parties $\mathcal{P} = \{P_1, \ldots, P_4\}$, divided into groups $\mathcal{G}_1, \ldots, \mathcal{G}_4$, where $\mathcal{G}_i = \mathcal{P} \setminus \{P_i\}$. Within each $\mathcal{G}_i$, the parties are arranged in circular order $P_{i+1}, P_{i+2}, P_{i+3}$, where $i + 1 = 1$, if $i = 4$ and so on. There exists a malicious adversary Adv, who can corrupt any 1 party. We consider two flavours of Adv: a computationally-unbounded Adv (for perfect-security) and a computationally-bounded Adv (for cryptographic-security). For cryptographically-secure protocols, we use κ and ssec to denote the computational and statistical-security parameter respectively. The parties are connected by pair-wise private and authentic channels. All the computations are performed over some finite field $\mathbb{F}$ and we assume that the parties want to compute a function f over $\mathbb{F}$, represented by a publicly known circuit cir over $\mathbb{F}$. For simplicity and without loss of generality, we assume that each party $P_i \in \mathcal{P}$ has a single input $x^{(i)}$ for f and there is a single function output $y = f(x^{(1)}, \ldots, x^{(4)})$, which is supposed to be learnt by all the parties. Apart from the input and output gates, cir consists of 2-input addition (linear) and multiplication gates, with c_M denoting the number of multiplication gates. The field size will vary, depending upon the MPC protocols. While for the perfectly-secure protocol we need $|\mathbb{F}| > 4$ to hold, for the cryptographically-secure protocols, the condition $|\mathbb{F}| \geq 2^{\mathsf{ssec}} \cdot 2c_M$ is imposed, to ensure that the correctness of the protocols hold except with probability at most $2^{-\mathsf{ssec}}$. We use the notation $[\ell]$ to denote the set $\{1, \ldots, \ell\}$.

Communication Model. We follow two different communication models, the completely *asynchronous* communication model of [8,20] with eventual message delivery and the *hybrid* communication setting of [5,7,51], which is a mix of synchronous and asynchronous communication models.

The asynchronous model does not put any restriction on the message delays and the only guarantee is that the messages of the honest participants are delivered *eventually*. The sequence of message delivery is controlled by a scheduler, which is under the control of the adversary. Due to the lack of any globally known upper bound on the message delays, no party can wait to receive communication from *all* its neighbours to avoid an endless wait (as a corrupt neighbour may not send any message) and hence in each step of an asynchronous protocol, a party can afford to receive communication from at most 3 parties (including itself), thus ignoring communication from 1 potentially honest neighbour.

The hybrid model is a combination of both synchronous, as well as asynchronous model. Namely, the system is assumed to be synchronous during the *initial* phase, followed by a completely asynchronous phase. Here the parties execute a protocol where the first r rounds are synchronous, with parties synchronized by a global clock and where the protocol operates as a sequence of rounds. In each such round, a party computes the set of messages to be sent to its neighbours, which are communicated over the channels, followed by receiving the messages sent by its neighbours to it in this round. There will be a publicly known upper bound on the message delays for the first r rounds and hence each party will know how long it has to wait for the messages from its neighbours for each of these r rounds. If an expected message does not arrive within this time-limit, then the receiving party can *substitute* a default message and proceed to the next round. Consequently, within the first r rounds, each party receives communication from *all* the parties. As in [7,51], during the synchronous phase, apart from the pair-wise channels, the network is augmented by a system-wide broadcast channel, modelled by an ideal-functionality $\mathcal{F}_{\mathsf{BC}}$, which allows any designated *sender* $\mathbf{S} \in \mathcal{P}$ to send some message identically to all the parties. Once the synchronous phase is over, the system becomes asynchronous and protocol execution occurs asynchronously. We assume $r = 2$ and $r = 1$ for our perfectly-secure and cryptographically-secure HMPC respectively.

Cryptographic Tools. To minimize the communication of our cryptographically-secure protocols, we assume a one-time symmetric-key set-up for a secure *pseudo-random function* (PRF) F [42] among the parties as follows.

- A random PRF key $k_{\mathcal{P}}$ among all the parties in $\mathcal{P}$.
- For every $\mathcal{G}_i$, a PRF key $k_{\mathcal{G}_i}$ among the parties in $\mathcal{G}_i$.

The above set-up implicitly implies that for *every* $\mathsf{D} \in \mathcal{P}$ and *every* $\mathcal{G}_i$, there is a common PRF key $k_{\mathsf{D},\mathcal{G}_i}$ among D and the parties in $\mathcal{G}_i$. If $\mathsf{D} \notin \mathcal{G}_i$, then $\mathsf{D} \cup \mathcal{G}_i = \mathcal{P}$ and hence $k_{\mathsf{D},\mathcal{G}_i} = k_{\mathcal{P}}$ holds. If $\mathsf{D} \in \mathcal{G}_i$, then $\mathsf{D} \cup \mathcal{G}_i = \mathcal{G}_i$ and hence $k_{\mathsf{D},\mathcal{G}_i} = k_{\mathcal{G}_i}$ holds. Using the appropriate common key, the respective parties can non-interactively generate common random values, whenever required, by running some appropriate secure mode of PRF. The above set-up can be made available, by running any standard MPC protocol. Since this is a one-time affair, we do not go into the exact details of the instantiation of the set-up. We also use a collision-resistant hash function $H(\cdot)$, with output size $|H|$ bits.

2.1 Definition and Existing Asynchronous Primitives

Definition 1. *A value $s \in \mathbb{F}$ is said to be $[[\cdot]]$-shared, if there exists $s_1, \ldots, s_4 \in \mathbb{F}$, with $s_1 + \ldots + s_4 = s$, such that for each $i \in [4]$, all (honest) parties in $\mathcal{G}_i$ hold s_i. The notation $[[s]]$ denotes the vector of shares $(s_1, \ldots, s_4)$. A set of values $S \in \mathbb{F}^\ell$ is $[[\cdot]]$-shared, if each $s^{(m)} \in S$ is $[[\cdot]]$-shared, where $[[s^{(m)}]] = (s_1^{(m)}, \ldots, s_4^{(m)})$.*

$[[\cdot]]$-sharings are linear: given $[[s^{(1)}]], [[s^{(2)}]]$ and public constants $c_1, c_2 \in \mathbb{F}$, then the parties can locally compute their shares corresponding to $[[c_1 \cdot s^{(1)} + c_2 \cdot s^{(2)}]]$.

In the *synchronous* setting, the standard security definition of MPC is based on the *universal composability* (UC) real-world/ideal-world based simulation paradigm [21]. Informally a protocol Π_{real} for MPC is defined to be secure in this paradigm, if it securely "emulates" what is called as an ideal-world protocol Π_{ideal}. In Π_{ideal}, all the parties give their respective inputs for the function f to be computed to a *trusted third party* (TTP), who locally computes the function output and sends it back to all the parties. Protocol Π_{real} is said to securely emulate Π_{ideal} if for any adversary attacking Π_{real}, there exists an adversary attacking Π_{ideal} that induces an indistinguishable *output* in Π_{ideal}, where the *output* is the concatenation of the outputs of the honest parties and the view of the adversary.

Extending the above definition to the asynchronous setting brings in a lot of additional technicalities, specially to deal with the eventual message delivery in the system, controlled by an adversarial scheduler. Due to this, most works in the domain of AMPC [38,39,5,50,28,51] follow a simpler "property based" security definition as originally proposed in [8,10]. Recently, in [29,31], asynchronous MPC has been modelled in the UC framework. Informally, in the case of the asynchronous setting, the local output of the honest parties is only an approximation of the pre-specified function f over a subset $\mathcal{C}$ of the local inputs of the parties, the rest being taken to be 0, where $|\mathcal{P} \setminus \mathcal{C}| = 3$ (this is to model the fact in a completely asynchronous setting, the inputs of all the parties can not be considered to avoid endless wait). Protocol Π_{real} is said to be secure in the asynchronous setting, if the local outputs of the honest players are correct (correctness), Π_{real} terminates eventually for all honest parties (termination) and the output of Π_{real} is indistinguishable from the output of Π_{ideal} (which involves a TTP that computes an approximation of f). If Π_{real} is executed in the hybrid communication model, then it is called an HMPC protocol. For the perfectly-secure setting, the termination property should be achieved with probability 1 and the output of Π_{real} should be perfectly-indistinguishable from that of Π_{ideal}. For the cryptographic setting, the output of Π_{real} should be computationally-indistinguishable from that of Π_{ideal} and a *negligible* error is allowed in the termination property.

As our main goal is to provide efficient AMPC and HMPC protocols, to avoid blurring the main focus of the paper and to avoid additional technicalities, we keep the formalism to a minimum in this extended abstract and defer the simulation-based security proofs of our protocols to the full version of the paper.

We stress that all AMPC protocols suffer from *input deprivation*, where inputs of up to t honest parties may be excluded from computation. A natural question is whether it is possible to achieve *input provision* and get the inputs of *all* the n parties for computation (i.e. $\mathcal{C} = \mathcal{P}$) in the hybrid setting. In [51] it is shown that any perfectly-secure HMPC protocol with 2 synchronous rounds suffers from input deprivation and so does our perfectly-secure HMPC protocol. On the other hand, our cryptographically-secure HMPC protocol achieves input provision, a property also achieved by the cryptographically-secure HMPC protocol of [7].

Asynchronous Broadcast and Agreement on a Common-subset. We use the asynchronous reliable broadcast (Acast) protocol of [19], which needs a communication of $57|H| + 54 + \frac{15}{2}|m|$ bits [46], to broadcast a message m consisting of $|m|$ bits. The protocol allows a sender $\mathbf{S} \in \mathcal{P}$ to send some message m identically to all the parties. If $\mathbf{S}$ is *honest*, then every honest party eventually terminates with output m. If $\mathbf{S}$ is *corrupt and* some honest party terminates with output $m^{\star}$, then eventually every honest party terminates with output $m^{\star}$. We say "*P_i acasts m*" to mean that $P_i \in \mathcal{P}$ acts as a $\mathbf{S}$ and invokes an instance of Acast protocol to broadcast m and the parties participate in this instance. Similarly, "*P_j receives $m^{\star}$ from the acast of P_i*" means that P_j terminates the instance of Acast protocol invoked by P_i as $\mathbf{S}$, with output $m^{\star}$.

In our asynchronous protocols, we come across situations, where each party is supposed to act as a dealer and share some values. Due to asynchronous communication, the parties cannot afford to wait for the termination of sharing instances of all the dealers, as the corrupt dealer need not invoke its instance. So the parties should terminate, immediately after terminating 3 out of the 4 sharing instances. However, each party may terminate a different subset of 3 sharing instances. The ACS protocol [8] allows agreement on a common subset of 3 sharing instances, which eventually terminate for all the honest parties.

Beaver's Circuit-Randomization [3]. Let $g(x, y, z)$ be a multiplication gate, such that the parties hold $[[x]]$ and $[[y]]$ and the goal is to compute a $[[\cdot]]$-sharing of $z = x \cdot y$. Moreover, let $([[u]], [[v]], [[w]])$ be a shared multiplication triple available with the parties, such that $w = u \cdot v$. We note that $z = (x - u + u) \cdot (y - v + v)$ and hence $z = (x - u) \cdot (y - v) + v \cdot (x - u) + u \cdot (y - v) + u \cdot v$. Based on this idea, to compute $[[z]]$, the parties first locally compute $[[d]] = [[x - u]] = [[x]] - [[u]]$ and $[[e]] = [[y - v]] = [[y]] - [[v]]$, followed by publicly reconstructing d and e. The parties then locally compute $[[z]] = d \cdot e + d \cdot [[v]] + e \cdot [[u]] + [[w]]$.

It is easy to see that if u and v are random and private, then during the above process, the view of the adversary remains independent of x and y. Namely, even after learning d and e, the privacy of the gate inputs and output is preserved. We denote this protocol as $\mathsf{Beaver}(([[x]], [[y]]), ([[u]], [[v]], [[w]]))$, which can be executed in a completely asynchronous setting. If the underlying public reconstruction protocol terminates for the honest parties, then the protocol Beaver eventually terminates for all the honest parties. We also note that if the auxiliary triple (u, v, w) is *not* a multiplication triple (namely $w = u \cdot v + \Delta$ for some non-zero Δ), then in the above protocol, $z = x \cdot y + \Delta$ holds.

Asynchronous Triple-transformation Protocol. We borrow the asynchronous triple-transformation protocol $\mathsf{TripTrans}$ from [28], which transforms a batch of independent secret-shared triples into another batch of secret-shared triples, which are related by some nice properties. The input is a set of $2k+1$ secret-shared triples $\{[[a^{(i)}]], [[b^{(i)}]], [[c^{(i)}]]\}_{i \in [2k+1]}$. The protocol outputs a set of $2k+1$ shared triples $\{[[A^{(i)}]], [[B^{(i)}]], [[C^{(i)}]]\}_{i \in [2k+1]}$, such that *all* the following hold:

- There exist polynomials $f_a(\cdot), f_b(\cdot)$ and $f_c(\cdot)$ of degree k, k and $2k$ respectively, such that $f_a(i) = A^{(i)}, f_b(i) = B^{(i)}$ and $f_c(i) = C^{(i)}$ hold for each $i \in [2k+1]$.
- For each $i \in [2k+1]$, the i^{th} output triple $(A^{(i)}, B^{(i)}, C^{(i)})$ is a multiplication triple, if and only if the i^{th} input triple $(a^{(i)}, b^{(i)}, c^{(i)})$ is a multiplication triple. This further implies that $f_c(\cdot) = f_a(\cdot) \cdot f_b(\cdot)$ holds if and only if all the $2k+1$ input triples are multiplication triples.
- Adv learns the i^{th} triple $(A^{(i)}, B^{(i)}, C^{(i)})$ iff it knows the i^{th} triple $(a^{(i)}, b^{(i)}, c^{(i)})$. This implies that if Adv knows k' input triples where $k' < (k+1)$, then adversary learns k' points on the $f_a(\cdot)$ and $f_b(\cdot)$ polynomials, leaving $(k+1)-k'$ "degree-of-freedom" on these polynomials. On the other hand, if $k' \geq (k+1)$, then Adv will completely know the $f_a(\cdot), f_b(\cdot)$ and $f_c(\cdot)$ polynomials.

The complexity of $\mathsf{TripTrans}$ is that of k instances of Beaver.

3 Perfectly-secure HMPC

3.1 Verifiable Secret-Sharing (VSS) with Party Elimination

Protocol Sh either allows a designated *dealer* $\mathsf{D} \in \mathcal{P}$ to verifiably generate a $[[\cdot]]$-sharing of its private input s or outputs a public dispute set $\mathcal{D}$ consisting of at most three parties, including the corrupt party. The privacy of s is preserved, if D is *honest*. Verifiability ensures that even if D is *corrupt* and the parties do not output a non-empty $\mathcal{D}$, then there exists some value, say $s^\star$, which is $[[\cdot]]$-shared.

During the first round, D distributes the $[[\cdot]]$-shares of s. To verify the consistency of shares, during the second round, the group members within each group $\mathcal{G}_i$ publicly exchange their respective versions of the received share; the public exchange is done by using the broadcast functionality. To maintain the privacy of the i^{th} share, the group members in $\mathcal{G}_i$ actually exchange a masked version of the received share, where the masks are pair-wise exchanged during the first round. At the end of the second round, the parties verify if there exists any $\mathcal{G}_i$, for which there exists a pair of parties P_j, P_k with inconsistency in their reported common share. This would imply that either D is corrupt or at least one among P_j, P_k is corrupt; consequently, the parties output the dispute set $\{\mathsf{D}, P_i, P_j\}$. Else, the parties output their respective shares.

3.2 Reconstruction Protocols

Protocol $\mathsf{RecPriv}$ allows a *designated* P_i to reconstruct a $[[\cdot]]$-shared value s, in a completely asynchronous setting. Since P_i will be a part of exactly 3 groups, it will have all the shares of s, except for the share s_i. So all the parties in $\mathcal{G}_i$ can send s_i to P_i, who waits to receive two identical copies of s_i, which eventually arrive, as there are at least 2 honest parties in $\mathcal{G}_i$. Upon receiving, P_i reconstructs s and terminates. Protocol RecPub allows *all* the parties in $\mathcal{P}$ to reconstruct s, by invoking 4 instances of $\mathsf{RecPriv}$, one on the behalf of each $P_i \in \mathcal{P}$.

3.3 Triple-sharing with Party Elimination

Protocol TripSh either allows a designated D to verifiably $[[\cdot]]$-share a multiplication triple (d, e, f) or outputs a public dispute set $\mathcal{D}$ containing at most three parties, with one of them being corrupt. The verifiability ensures that if the parties output $([[d]], [[e]], [[f]])$, then $f = d \cdot e$ holds, even if D is *corrupt*. Moreover, for an *honest* D, the view of adversary will be independent of (d, e, f). The protocol is divided into two phases, a synchronous phase for the first two rounds, followed by an asynchronous phase. During the synchronous phase, D $[[\cdot]]$-shares 3 random multiplication triples, by executing instances of Sh. Independently, each P_i $[[\cdot]]$-shares a random *verifying* multiplication triple by executing Sh. If a non-empty $\mathcal{D}$ is identified during any of these Sh instances, then the parties terminate the protocol with $\mathcal{D}$. Else, they proceed to the asynchronous phase.

The goal of the asynchronous phase is to verify if D's triples are multiplication triples. For this, D's triples $\{[[a^{(i)}]], [[b^{(i)}]], [[c^{(i)}]]\}_{i \in [3]}$ are transformed into the triples $\{[[A^{(i)}]], [[B^{(i)}]], [[C^{(i)}]]\}_{i \in [3]}$, by executing $\mathsf{TripTrans}$ with $k = 1$. Let $f_a(\cdot)$, $f_b(\cdot)$ and $f_c(\cdot)$ be the corresponding polynomials of degree $1, 1$ and 2 respectively, which are guaranteed to exist at the end of $\mathsf{TripTrans}$. It follows that $\{[[a^{(i)}]], [[b^{(i)}]], [[c^{(i)}]]\}_{i \in [3]}$ are multiplication triples, iff $f_c(\cdot) = f_a(\cdot) \cdot f_b(\cdot)$ holds and the parties proceed to verify the same. To verify if $f_c(\cdot) \overset{?}{=} f_a(\cdot) \cdot f_b(\cdot)$, the parties *asynchronously* test if $(f_a(i), f_b(i), f_c(i))$ is a multiplication triple, using the "help" of P_i, for each $i \in [4]$, where *only* P_i learns $(f_a(i), f_b(i), f_c(i))$, but everyone learns the outcome of the test. It will be ensured that if P_i is *honest* and if the test is positive, then $(f_a(i), f_b(i), f_c(i))$ is a multiplication triple. But a negative test outcome occurs, only if either D or P_i is corrupt, in which case, the parties output $\mathcal{D} = \{\mathsf{D}, P_i\}$. Consequently, if no $\mathcal{D}$ is obtained during the asynchronous phase, then it confirms that $f_c(i) = f_a(i) \cdot f_b(i)$ holds, corresponding to *all honest* parties P_i, further implying $f_c(\cdot) = f_a(\cdot) \cdot f_b(\cdot)$, as the degree of $f_a(\cdot), f_b(\cdot)$ and $f_c(\cdot)$ are $1, 1$ and 2 respectively. If D is *honest* and no disputes are obtained, then throughout adversary learns only $(f_a(i), f_b(i), f_c(i))$ corresponding to corrupt P_i, leaving "one degree of freedom" in these polynomials. Hence in this case, the parties output a $[[\cdot]]$-sharing of the multiplication triple $f_a(\beta), f_b(\beta)$ and $f_c(\beta)$ on the *behalf* of D, where β is different from $1, \ldots, 4$.

The verification of $f_c(\cdot) \overset{?}{=} f_a(\cdot) \cdot f_b(\cdot)$ is done as follows. At the end of $\mathsf{TripTrans}$, the parties have $[[A^{(i)} = f_a(i)]]_{i \in [3]}, [[B^{(i)} = f_b(i)]]_{i \in [3]}$ and $[[C^{(i)} = f_c(i)]]_{i \in [3]}$. The parties locally compute $[[A^{(4)} = f_a(4)]], [[B^{(4)} = f_b(4)]]$ and $[[C^{(4)} = f_c(4)]]$. Each $([[A^{(i)}]], [[B^{(i)}]], [[C^{(i)}]])$ is then verified for the relation $C^{(i)} \overset{?}{=} A^{(i)} \cdot B^{(i)}$, using the verifying triple $([[X^{(i)}]], [[Y^{(i)}]], [[Z^{(i)}]])$, shared by P_i during the synchronous phase. Namely, the parties recompute $[[\overline{C}^{(i)} = A^{(i)} \cdot B^{(i)}]]$ from

$[[A^{(i)}]]$ and $[[B^{(i)}]]$, by executing Beaver, treating $([[X^{(i)}]], [[Y^{(i)}]], [[Z^{(i)}]])$ as an auxiliary triple. Then the parties compute the difference $[[D^{(i)}]] = [[\overline{C}^{(i)}]] - [[C^{(i)}]]$ and *publicly* reconstruct $D^{(i)}$, to check if it is zero. If $D^{(i)}$ is found to be non-zero, then the parties output $\mathcal{D} = \{D, P_i\}$. Clearly, if D, P_i are *honest*, then $D^{(i)} = 0$. Moreover, the privacy of $([[A^{(i)}]], [[B^{(i)}]], [[C^{(i)}]])$ is maintained, as the auxiliary multiple triple will be random and private.

3.4 Triple Generation with Party Elimination

Protocol $\mathsf{TripGen}$ outputs either a dispute set $\mathcal{D}$, containing the corrupt party, or a $[[\cdot]]$-sharing of a random and private multiplication triple. In the protocol, each $P_i \in \{P_1, P_2, P_3\}$ invokes an instance of TripSh to share a random multiplication triple $(d^{(i)}, e^{(i)}, f^{(i)})$. The parties terminate, if a non-empty $\mathcal{D}$ is obtained in any of the TripSh instance. Else the parties have 3 shared *multiplication triples* $\{(d^{(i)}, e^{(i)}, f^{(i)})\}_{i \in [3]}$, with at least two of them being shared by honest parties, which are random and private. Since the exact identity of the honest parties are not known, the parties securely and *asynchronously* "extract" out a random and private multiplication triple (u, v, w) from $\{(d^{(i)}, e^{(i)}, f^{(i)})\}_{i \in [3]}$ as follows.

The parties transform the shared triples $\{(d^{(i)}, e^{(i)}, f^{(i)})\}_{i \in [3]}$ into another set of shared triples $\{(U^{(i)}, V^{(i)}, W^{(i)})\}_{i \in [3]}$ by executing an instance of $\mathsf{TripTrans}$ with $k = 1$. Let $f_U(\cdot), f_V(\cdot)$ and $f_W(\cdot)$ be the resultant polynomials of degree at most $1, 1$ and 2 respectively, such that $U^{(i)} = f_U(i), V^{(i)} = f_V(i)$ and $W^{(i)} = f_W(i)$. Since all the input triples $\{(d^{(i)}, e^{(i)}, f^{(i)})\}_{i \in [3]}$ for $\mathsf{TripTrans}$ are multiplication triples, it follows that the condition $f_W(\cdot) = f_U(\cdot) \cdot f_V(\cdot)$ holds. Additionally, adversary will know at most one value on these polynomials, as it will know at most one of the input triples. Consequently, from the view point of the adversary, there exists one degree of freedom in these polynomials and hence the parties set (u, v, w) to be the value of these polynomials at β.

3.5 Perfectly-secure Hybrid MPC Protocol

Protocol $\mathsf{PerfMPC}$ starts with a triple-generation and input phase, executed in parallel, followed by a circuit-evaluation phase. The input phase consumes the first 2 synchronous rounds and so does the triple-generation phase, which continues to the asynchronous phase after the consumption of the first 2 synchronous rounds. The circuit-evaluation is completely asynchronous. During the triple-generation phase, the parties execute $\mathsf{TripGen}$ c_M number of times and try to generate c_M number of shared and random multiplication triples. On the other hand, during the input phase, each party acts as a dealer and $[[\cdot]]$-shares its input for cir by executing an instance of Sh. The parties then wait for the termination of the instances of $\mathsf{TripGen}$ and Sh. There are two possible cases. If a non-empty dispute set $\mathcal{D}$ is obtained during any of the instances of $\mathsf{TripGen}$ or Sh, then the parties designate the first party P_{hon} from the set $\mathcal{P} \setminus \mathcal{D}$ to perform the circuit evaluation in *clear* during the circuit-evaluation phase. If there are multiple instances where a non-empty $\mathcal{D}$ is obtained, then the parties focus on the first such instance, ensuring that all the honest parties agree on a common $\mathcal{D}$. Note that P_{hon} will be non-empty, as $\mathcal{D}$ will have at most 3 parties and since it will always include the corrupt party, P_{hon} will be honest. Hence, during the circuit-evaluation phase, all the parties provide their input for cir to P_{hon}. Since the circuit-evaluation happens asynchronously, P_{hon} waits for the input from 3 parties and substitutes 0 as the missing input and proceeds to evaluate cir with these inputs. Hence, the protocol does not ensure input-provision in this case, as the missing party could be a potentially slow honest party, whose actual input is substituted with 0. The privacy of the inputs of the honest parties whose inputs are considered for the computation is preserved, since P_{hon} is *honest*.

If no $\mathcal{D}$ is obtained during the triple-generation phase or input phase, then the parties proceed to perform shared circuit-evaluation, using the shared random multiplication triples from the triple-generation phase and the shared inputs from the input phase. During the shared circuit-evaluation, the parties evaluate each gate of the circuit, maintaining the invariant that if the inputs of the gate are available in a $[[\cdot]]$-shared fashion, then the output of the gate is also available in a $[[\cdot]]$-shared fashion. While maintaining the invariant for the addition gate requires only local computation, for the multiplication gates, protocol Beaver is deployed, using an auxiliary multiplication triple from the triple-generation phase. Finally, once the circuit output is available in a $[[\cdot]]$-shared fashion, the parties publicly reconstruct the same. Note that in this case input-provision is ensured, as the actual input of all honest parties are considered for the computation.

4 Cryptographically-secure HMPC and AMPC

We take the leverage of the PRF key set-up and hash function, to reduce the communication complexity of the building blocks used in our perfectly-secure protocol. All the building blocks are *robust*, with no party elimination.

4.1 Cryptographically-secure Synchronous VSS

Protocol CSh allows a dealer D to verifiably $[[\cdot]]$-share its input. In the protocol, the shares for $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$ are computed non-interactively, using the appropriate common randomness; this ensures that none of these groups are inconsistent. The share for $\mathcal{G}_4$ is computed by D and a masked version of it is broadcast by D, where the mask is computed using the common randomness of D and $\mathcal{G}_4$. The parties in $\mathcal{G}_4$ on receiving the masked share, unmask it, thus ensuring that $\mathcal{G}_4$ is also not inconsistent. The privacy for an *honest* D is maintained, as there exists at least one $\mathcal{G}_i$ consisting of *only* honest parties, such that the corresponding s_i will be unknown because either it is locally generated by D and $\mathcal{G}_i$ or available in a masked fashion, with the mask being known only to D and $\mathcal{G}_i$.

4.2 Reconstruction Protocols with Cryptographic Security

Protocol $\mathsf{RecPriv}$ is modified to get $\mathsf{CRecPriv}$, where *only* the first two parties in $\mathcal{G}_i$ send the share s_i to the designated party P_i, while the third party in $\mathcal{G}_i$

sends a hash of s_i. Party P_i either waits for two identical copies of s_i or a copy of s_i and a "matching" hash value. Extending this idea for the case when there exist ℓ $[[\cdot]]$-shared values to be reconstructed by P_i, allows to almost avoid the communication of the shares from one of the members of $\mathcal{G}_i$. Namely, only the first two members of $\mathcal{G}_i$ has to send all the s_i shares, while the third member sends a *single* hash value for all the s_i shares. Executing $\mathsf{CRecPriv}$ once for each P_i, we get the public reconstruction protocol $\mathsf{CRecPub}$.

4.3 Cryptographically-secure Hybrid Triple-sharing Protocol

Protocol $\mathsf{CTripSh}$ outputs ℓ multiplication triples. The protocol is similar to TripSh, with the following differences. Instead of sharing only 3 multiplication triples, D shares $2\ell+1$ multiplication triples by executing instances of CSh, which ensures that these triples are robustly $[[\cdot]]$-shared by D. However, no verifying-triples need to be shared by the individual parties, as D's triples are verified for the multiplication relationship *probabilistically.* Namely, D's triples are transformed by executing an instance of $\mathsf{TripTrans}$ and fitting polynomials $f_a(\cdot), f_b(\cdot)$ and $f_c(\cdot)$ of degree ℓ, ℓ and 2ℓ respectively through the transformed triples.

The parties next publicly check if $f_c(r) \overset{?}{=} f_a(r) \cdot f_b(r)$ holds, for a randomly chosen r, to verify D's shared triples. The idea here is that if a *corrupt* D has not shared multiplication triples, then the test $f_c(r) \overset{?}{=} f_a(r) \cdot f_b(r)$ will fail for a randomly chosen r not known in advance to D, except for the case when r is a root of the polynomial $f_c(\cdot) - f_a(\cdot) \cdot f_b(\cdot)$, which happens with probability at most $\frac{2\ell}{|\mathbb{F}|}$. If D is *honest*, then the test is always successful and throughout the protocol the adversary learns only one point on the polynomials, namely $(f_a(r), f_b(r), f_c(r))$, leaving ℓ degrees of freedom in the polynomials. Consequently, if the test is successful, the parties output a $[[\cdot]]$-sharing of ℓ publicly known distinct points on the polynomials (different from r and the $2\ell+1$ values used during $\mathsf{TripTrans}$) on behalf of D. For this, we need $|\mathbb{F}| > 3\ell$ to hold, which is the case, as ℓ will be c_M in our MPC protocol and $2^{\mathsf{ssec}} \cdot 2c_M > 3c_M$ holds for any $\mathsf{ssec} \geq 1$.

4.4 Cryptographically-secure Hybrid Triple-Generation

Protocol $\mathsf{CTripGen}$ is similar to the perfectly-secure protocol $\mathsf{TripGen}$, with the following modifications. Each $P_i \in \{P_1, P_2, P_3\}$ now verifiably $[[\cdot]]$-shares ℓ number of random and private multiplication triples by executing a *single* instance of $\mathsf{CTripSh}$. The parties then output ℓ random multiplication triples by grouping the triples of P_1, P_2, P_3 into ℓ batches and then securely extracting out one random multiplication triple from each batch. For the sake of efficiency, a common random r can be used for verifying the multiplication triples in all the 3 instances of $\mathsf{CTripSh}$. Similarly, whenever possible, all the values which are supposed to be publicly reconstructed across all the instances of $\mathsf{TripTrans}$ can be "clubbed" together and reconstructed by a *single* instance of $\mathsf{CRecPub}$. Conditioned on the event that the triples shared by P_1, P_2, P_3 are multiplication triples, which occurs except with probability $\frac{2\ell}{|\mathbb{F}|}$, the output triples are multiplication triples.

4.5 Cryptographically-secure HMPC

In the protocol, the parties generate c_M shared triples by executing $\mathsf{CTripGen}$; simultaneously each party $[[\cdot]]$-shares its input by invoking CSh. So the first synchronous round is used for *both* generating the triples, as well as for getting the inputs of *all* the parties, thus *guaranteeing* input-provision. This is followed by shared circuit-evaluation and reconstruction of the shared circuit output.

4.6 Cryptographically-secure AMPC

The cryptographically-secure AMPC is similar to the HMPC protocol. The difference is that the protocol CSh is now executed *asynchronously*, where the parties do not have access to $\mathcal{F}_{\mathsf{BC}}$. Consequently, D acasts m_4 and the parties wait to terminate this acast instance. We call this modified protocol as ASh, which terminates *only if* D acasts m_4 (which an *honest* D eventually does). All the instances of CSh in the HMPC protocol are now replaced by ASh. As different honest parties can terminate various ASh instances in arbitrary order, an instance of ACS is executed during the pre-processing stage to agree on the ASh instances to be considered during the triple-generation protocol. Similarly, an instance of ACS is required to agree on the set of input providers for the circuit evaluation. The details are deferred to the full version of the paper.

5 Experimental Results

Here we evaluate our implementation of the cryptographically-secure AMPC (section 4.6) with an emphasis on the communication and computation costs.

Network and Hardware Details: We run our experiments on Microsoft Azure Standard D1 v2 instances comprising 1 vcpu and 3.5 GB RAM. We benchmark over both LAN and WAN networks. For LAN, we use four instances in the same region (Southeast Asia), where the average bandwidth is 671 Mbps while the average round trip time (RTT) is 906.88 ± 345.38 μs as measured by the *iperf* and *irtt* tools respectively. For WAN, the cloud instances are in different regions (Southeast Asia, Northern Europe, Central US, Brazil South) with an average bandwidth of 141 Mbps and an average RTT of 182.42 ± 0.34 ms.

Software Details: HoneyBadgerMPC [45] is a MPC-based confidentiality layer for blockchains implemented in Python and provides many AMPC primitives like ACast and ACS. We thus implement our protocol in Python 3 using HoneyBadgerMPC [45] and SageMath [56] for field and matrix computations. Though single-threaded, it makes use of asynchronous I/O operations using the *asyncio* library. Data is serialized before communication using the *pickle* library which was observed to have an average overhead of 1.32 times a fixed length encoding when serializing a list of integers. In the experiments discussed below, all parties behave honestly unless stated otherwise. Each experiment has been run 50 times and the data reported below is the average across all runs.

5.1 Primitives

We use AES-128 and SHA256 for instantiating the PRF F and the hash function H respectively. The PRF is run in counter mode to generate a stream of pseudorandom values. Asynchronous broadcast (ACast) is instantiated using the Reliable Broadcast protocol described in [46] which improves the efficiency of Bracha's broadcast [15] using erasure codes [19]. Asynchronous Common Subset (ACS) is instantiated using the protocol proposed by Ben-Or et al. [10]. We adopt the implementation of these primitives from HoneyBadgerMPC.

The main computational bottleneck for triple-generation is polynomial interpolation, since the degrees of polynomials is proportional to the number of multiplication gates. We opt for the efficient Fast Fourier Transform (FFT) based interpolation, which runs in $\mathcal{O}(m \log m)$ time, where m is the number of points over which the polynomial is interpolated, and allows interpolation only when $m + 1$ is a power of 2. The requirement on m essentially means that we can generate triplets only in batches of powers of 2. Thus any arbitrary l number of triples can be generated in $\log l$ batches. Computation is done over the field $\mathbb{Z}_p$ where $p = 2^{64} - 1835007$ is a 64-bit prime number.

5.2 Triple-Generation Phase

As discussed in Section 4.6, the triple-generation phase involves the generation of l random and secret-shared multiplication triples. Since these triples are consumed in the circuit-evaluation phase, during the evaluation of multiplication gates, l is usually equal to or a multiple of c_M, the number of multiplication gates in the circuit. Table 1 summarises the triple-generation throughput (namely the number of triples generated per second) and communication costs in the LAN and WAN settings for different values of l, the number of triples generated.

The throughput initially increases with l since the bandwidth is not completely utilized. However in LAN, the throughput starts to decrease for large values of l since the computation costs dominate. On the other hand, the throughput continues to increase in WAN, since network latency is the main bottleneck.

Table 1: Throughput in triples per second and communication in bytes per triple for the triple-generation protocol across different values of l in LAN and WAN settings.

l	LAN (triples/s)	WAN (triples/s)	Comm. (B/triple)
31	100.08	8.92	869.26
63	125.85	17.22	695.59
127	140.15	31.23	610.48
255	138.20	51.72	564.35
511	136.67	71.69	547.47
1023	132.36	88.42	527.05
2047	126.60	100.04	515.08

Table 2: Throughput in triples per second and communication in Bytes per triple for generating $\approx 10^4$ triples with different batch sizes.

l	b	LAN (triples/s)	WAN (triples/s)	Comm. (B/triple)
31	323	178.72	162.74	550.78
63	159	169.74	155.40	539.70
127	79	158.62	148.06	534.32
255	40	151.70	141.28	531.63
511	20	144.65	135.14	530.28
1023	10	135.93	126.62	529.62
2047	5	126.94	118.17	529.28

Trade-off between Computation and Communication: As l becomes increasingly large, the interpolation is over polynomials of large degrees and the computation costs start to dominate leading to lower throughput. Running b instances of the triple-generation protocol in parallel for smaller values of l allows us to decrease the computation time with a linear overhead in b for the communication cost. Computing l' multiplication triples requires each honest party to secret share $2l' + 1$ local random triples. Thus, running b instances of the protocol with $l = \frac{l'}{b}$ in parallel requires each party to secret share an additional $b - 1$ local triples. Similarly the communication grows linearly with b in case of triple-transformation and verification of shared triples.

Table 2 summarizes the throughput and communication costs when generating around 10^4 triples using different batch sizes. In case of both LAN and WAN settings, we achieve the highest throughput when the batch size $b = 31$. As expected, the communication cost is also the highest in this case though we achieve a 41% increase in throughput at the cost of a 4% increase in communication.

5.3 Secure Prediction

We benchmark our protocol for secure prediction on a linear regression model for the MNIST dataset [43] with $d = 784$ features. Given secret shares of the weight vector $[[\overrightarrow{\mathbf{w}}]] = ([[w_1]], [[w_2]] \dots [[w_d]])$, the bias term $[[b]]$ and the input vector $[[\overrightarrow{\mathbf{x}}]] = ([[x_1]], [[x_2]], \dots [[x_d]])$, secure prediction on a linear regression model requires securely computing $[[\overrightarrow{\mathbf{w}} \odot \overrightarrow{\mathbf{x}} + b]]$ where $\odot$ denotes vector dot product. A decimal value $\widetilde{v}$ is encoded as an element in $\mathbb{Z}_p$ as $v = \lfloor \widetilde{v} 2^f \rfloor \mod p$ where we set $f = 13$. Thus, decimal numbers are encoded as fixed point numbers with $f = 13$ bits for the fractional part and $63 - 13 - 1 = 49$ bits for integer part (we cannot use all 50 bits for the integer part since we work with a 64-bit prime field). Since the linear regression function has a multiplicative depth of 1, we have 26 bits for the fractional part and 36 bits for the integer part in the output and bias term which provides sufficient accuracy. Thus, we do not need additional mechanisms for secure fixed point arithmetic.

Table 3 summarizes the performance of our protocol for secure prediction on linear regression on the MNIST dataset while Figure 1 compares the circuit-evaluation throughput against the number of features.

Robustness to network delays: As asynchronous protocols have the advantage of running at the actual speed of the underlying network, we evaluate this property by randomly delaying the messages sent by a party up to 100ms and 1s in the LAN and WAN setting respectively. We affect each party with such random delays incrementally across experiments, the results for which are summarized in Figure 2. We find that there is negligible change in latency in both the LAN and WAN settings when a single party is affected since the asynchronous protocol requires communication from at most 3 parties at each step. The latency increases as more number of parties are affected since the RTT increases. However, the latency is proportional to the average RTT rather than the RTT for any single party as shown by the gradual increase in latency with the number

Table 3: Latency in seconds and communication cost in bytes for the triple-generation and circuit-evaluation protocols for secure prediction on a linear regression model for the MNIST dataset.

Setting	Triple Gen.	Circuit Eval.
LAN (s)	4.86	0.17
WAN (s)	8.48	0.27
Comm. (KiB)	437.73	32.23

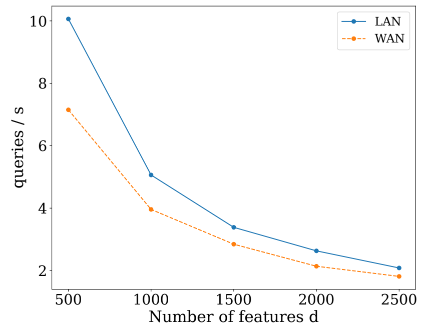
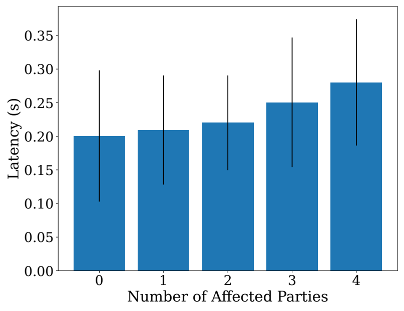
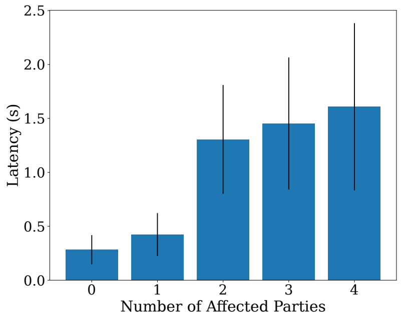


Fig. 1: Circuit-evaluation throughput of linear regression against the number of features in the dataset.

of affected parties. In a synchronous protocol, even a single affected party would require setting the global round delay Δ to the highest possible value of 100ms in the LAN setting and 1s in the WAN setting.



(a) LAN: The messages sent by affected parties are delayed up to 100ms.



(b) WAN: The messages sent by affected parties are delayed up to 1s.

Fig. 2: Latency in LAN and WAN setting vs the number of slow affected parties.

Acknowledgement: We would like to thank the anonymous reviewers of INDOCRYPT 2020 for several helpful suggestions and comments.

References

1. T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In *Symposium on Security and Privacy*, pages 843–862. IEEE Computer Society, 2017.

2. T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *CCS*, pages 805–817. ACM, 2016.
3. D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.
4. D. Beaver, S. Micali, and P. Rogaway. The Round Complexity of Secure Protocols (Extended Abstract). In *STOC*, pages 503–513. ACM, 1990.
5. Z. Beerliová-Trubíniová and M. Hirt. Simple and Efficient Perfectly-Secure Asynchronous MPC. In *ASIACRYPT*, volume 4833 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 2007.
6. Z. Beerliová-Trubíniová and M. Hirt. Perfectly-Secure MPC with Linear Communication Complexity. In *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2008.
7. Z. Beerliová-Trubíniová, M. Hirt, and J. B. Nielsen. On the Theoretical Gap between Synchronous and Asynchronous MPC Protocols. In *PODC*, pages 211–218. ACM, 2010.
8. M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous Secure Computation. In *STOC*, pages 52–61. ACM, 1993.
9. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *STOC*, pages 1–10. ACM, 1988.
10. M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous Secure Computations with Optimal Resilience (Extended Abstract). In *PODC*, pages 183–192. ACM, 1994.
11. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.
12. D. Bogdanov, R. Talviste, and J. Willemson. Deploying Secure Multi-Party Computation for Financial Data Analysis - (Short Paper). In *FC*, volume 7397 of *Lecture Notes in Computer Science*, pages 57–64. Springer, 2012.
13. P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure Multiparty Computation Goes Live. In *FC*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.
14. E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs. In *CCS*, pages 869–886. ACM, 2019.
15. G. Bracha. An Asynchronous [(n-1)/3]-Resilient Consensus Protocol. In *PODC*, pages 154–162. ACM, 1984.
16. M. Byali, H. Chaudhari, A. Patra, and A. Suresh. FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning. *IACR Cryptology ePrint Archive*, 2019:1365, 2019.
17. M. Byali, C. Hazay, A. Patra, and S. Singla. Fast Actively Secure Five-Party Computation with Security Beyond Abort. In *CCS*, pages 1573–1590. ACM, 2019.
18. M. Byali, A. Joseph, A. Patra, and D. Ravi. Fast Secure Computation for Small Population over the Internet. In *CCS*, pages 677–694. ACM, 2018.
19. C. Cachin and S. Tessaro. Asynchronous Verifiable Information Dispersal. In *SRDS*, pages 191–202. IEEE Computer Society, 2005.
20. R. Canetti. *Studies in Secure Multiparty Computation and Applications.* PhD thesis, Weizmann Institute, Israel, 1995.

21. R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptology*, 13(1):143–202, 2000.
22. N. Chandran, J. A. Garay, P. Mohassel, and S. Vusirikala. Efficient, Constant-Round and Actively Secure MPC: Beyond the Three-Party Case. In *CCS*, pages 277–294. ACM, 2017.
23. N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning. In *European Symposium on Security and Privacy*, pages 496–511. IEEE, 2019.
24. H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *CCSW@CCS*, pages 81–92. ACM, 2019.
25. K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *CRYPTO*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64. Springer, 2018.
26. A. Choudhury, J. Loftus, E. Orsini, A. Patra, and N. P. Smart. Between a Rock and a Hard Place: Interpolating between MPC and FHE. In *ASIACRYPT*, volume 8270 of *Lecture Notes in Computer Science*, pages 221–240. Springer, 2013.
27. A. Choudhury and A. Patra. Optimally Resilient Asynchronous MPC with Linear Communication Complexity. In *ICDCN*, pages 5:1–5:10. ACM, 2015.
28. A. Choudhury and A. Patra. An Efficient Framework for Unconditionally Secure Multiparty Computation. *IEEE Trans. Information Theory*, 63(1):428–468, 2017.
29. R. Cohen. Asynchronous Secure Multiparty Computation in Constant Time. In *PKC*, volume 9615 of *Lecture Notes in Computer Science*, pages 183–207. Springer, 2016.
30. R. Cohen and Y. Lindell. Fairness versus Guaranteed Output Delivery in Secure Multiparty Computation. In *ASIACRYPT*, volume 8874 of *Lecture Notes in Computer Science*, pages 466–485. Springer, 2014.
31. S. Coretti, J. A. Garay, M. Hirt, and V. Zikas. Constant-Round Asynchronous Multi-Party Computation Based on One-Way Functions. In *ASIACRYPT*, volume 10032 of *Lecture Notes in Computer Science*, pages 998–1021, 2016.
32. I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In *PKC*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.
33. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.
34. J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *EUROCRYPT*, volume 10211 of *Lecture Notes in Computer Science*, pages 225–255, 2017.
35. C. Gentry, S. Halevi, and V. Vaikuntanathan. A Simple BGN-Type Cryptosystem from LWE. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 506–522. Springer, 2010.
36. O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.
37. M. Hirt and U. M. Maurer. Robustness for Free in Unconditional Multi-party Computation. In *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 101–118. Springer, 2001.
38. M. Hirt, J. B. Nielsen, and B. Przydatek. Cryptographic Asynchronous Multi-party Computation with Optimal Resilience (Extended Abstract). In *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 322–340. Springer, 2005.

39. M. Hirt, J. B. Nielsen, and B. Przydatek. Asynchronous Multi-Party Computation with Quadratic Communication. In *ICALP*, volume 5126 of *Lecture Notes in Computer Science*, pages 473–485. Springer, 2008.
40. Y. Ishai, R. Kumaresan, E. Kushilevitz, and A. Paskin-Cherniavsky. Secure Computation with Minimal Interaction, Revisited. In *CRYPTO*, volume 9216 of *Lecture Notes in Computer Science*, pages 359–378. Springer, 2015.
41. J. Katz, V. Kolesnikov, and X. Wang. Improved Non-Interactive Zero Knowledge with Applications to Post-Quantum Signatures. In *CCS*, pages 525–537. ACM, 2018.
42. J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.
43. Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.
44. Y. Lindell. Secure Multiparty Computation (MPC). Cryptology ePrint Archive, Report 2020/300, 2020.
45. D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. K. Miller. HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and its Application to Anonymous Communication. In *CCS*, pages 887–903. ACM, 2019.
46. A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The Honey Badger of BFT Protocols. In *CCS*, pages 31–42. ACM, 2016.
47. P. Mohassel and P. Rindal. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS*, pages 35–52. ACM, 2018.
48. P. Mohassel, M. Rosulek, and Y. Zhang. Fast and Secure Three-party Computation: The Garbled Circuit Approach. In *CCS*, pages 591–602. ACM, 2015.
49. P. Mohassel and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society, 2017.
50. A. Patra, A. Choudhury, and C. Pandu Rangan. Efficient Asynchronous Verifiable Secret Sharing and Multiparty Computation. *J. Cryptology*, 28(1):49–109, 2015.
51. A. Patra and D. Ravi. On the Power of Hybrid Networks in Multi-Party Computation. *IEEE Trans. Information Theory*, 64(6):4207–4227, 2018.
52. A. Patra and A. Suresh. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. *IACR Cryptology ePrint Archive*, 2020:42, 2020.
53. T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract). In *STOC*, pages 73–85. ACM, 1989.
54. R. Rachuri and A. Suresh. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. *IACR Cryptology ePrint Archive*, 2019:1315, 2019.
55. A. Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.
56. The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.9)*, 2020. `https://www.sagemath.org`.
57. S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. *PoPETs*, 2019(3):26–49, 2019.
58. A. C. Yao. Protocols for Secure Computations (Extended Abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.