Efficient Secure Computation Enabled by Blockchain Technology

by

Guy Zyskind

Submitted to the Program in Media Arts and Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Program in Media Arts and Sciences
May 6, 2016

Certified by .....
Alex 'Sandy' Pentland
Toshiba Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by .....
Pattie Maes
Academic Head
Program in Media Arts and Sciences

Efficient Secure Computation Enabled by Blockchain Technology

by

Guy Zyskind

Submitted to the Program in Media Arts and Sciences
on May 6, 2016, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

For several decades, secure multiparty computation has been the topic of extensive research, as it enables computing any functionality in a privacy-preserving manner, while ensuring correctness of the outputs. In recent years, the field has seen tremendous progress in terms of efficiency, although most results remained impractical for real applications concerning complex functionalities or significant data.

When privacy is not a concern and we are only interested in achieving consensus in a distributed computing environment, the rise of cryptocurrencies, specifically Bitcoin, has presented an efficient and robust solution that exceeds the limits imposed by prior theoretical results. Primarily, Bitcoin's relative efficiency and superiority in achieving consensus is due to its inclusion of incentives. By doing so, it extends the standard cryptographic model to one that reasons about security through rationality of the different players.

Inspired by this idea, this thesis focuses on the development of an efficient, general-purpose secure computation platform that relies on blockchain and cryptocurrencies (e.g., Bitcoin) for efficiency and scalability. Similar to how Bitcoin transformed the idea of distributed consensus, the goal in this work is to take secure multi-party computation from the realm of theory to practice. To that end, a formal model of secure computation in an environment of rational players is developed and is used to show how in this framework, efficiency is improved compared to the standard cryptographic model.

The second part of this thesis deals with improving secure computation protocols over the integers and fixed-point numbers. The protocols and tools developed are a significant improvement over the current state-of-the-art, with an optimally efficient secure comparison protocol (for up to 64-bit integers) and better asymptotic bounds for fixed-point division.

Thesis Supervisor: Alex 'Sandy' Pentland
Title: Toshiba Professor of Media Arts and Sciences

Acknowledgments

I would like to thank my advisor, Alex 'Sandy' Pentland, for his continuous guidance and support. I will always be grateful for his kind mentorship, stretching beyond his role as an advisor. Without his counsel, this thesis would not have been possible, and I am fortunate to have had the opportunity to learn from his experience. I would also like to thank my thesis readers – Srini Devadas and Andrew W. Lo, for their helpful comments, reviews and discussions, allowing me to improve this work. Their support has enabled me to successfully navigate through the complex realms of distributed systems, economics and finance.

To my family, who has selflessly helped me through countless of challenges, words cannot describe my debt of gratitude to you. None of this would have happened without the love, wisdom and support you have unconditionally bestowed upon me throughout my life.

To my wife Rinat, I am thankful for the courage to take this wonderful adventure with me, and for enduring all the hardships we encountered along the way. For believing in my abilities even when my own confidence wavered, and for being my beacon of hope in these past (challenging) years. To my parents, Yossi and Orna, my brother Nir and my sister Einat - I would like to thank for shaping me into the person I am today. For always making sure I stay on course, and for guiding me to my own *De'rech Ha'melech*. Last but not least, I thank my dear beagle Newton, who has taught me that the best solutions come to you when you walk your dog.

Efficient Secure Computation Enabled by Blockchain Technology
by
Guy Zyskind

Thesis Advisor .....
Alex 'Sandy' Pentland
Toshiba Professor of Media Arts and Sciences
MIT Program in Media Arts and Sciences

Thesis Reader .....
Andrew W. Lo
Charles E. and Susan T. Harris Professor of Finance
MIT Sloan School of Management

Thesis Reader .....
Srini Devadas
Edwin Sibley Webster Professor of Electrical Engineering and Computer Science
MIT Computer Science and Artificial Intelligence Laboratory

Contents

1 Introduction

The field of secure computation explores solutions for analyzing data while keeping the inputs private at all times. Although in appearance, this sounds contradictory in nature, completeness theorems have shown that solutions exist not only for specific functions of interest, but to all of them [6]. Secure computation was originally formulated for the two-party case by Yao [2], who has presented a solution in the form of decentralization. The original formulation asked the question – how can two mutually distrusting parties collectively compute an answer to a question, without each party having to disclose their private information? Generalized solutions for the n-party case quickly followed. An interesting model that was developed later is that of separating the clients from the servers. In this model, clients can store their data with a group of parties (a peer-to-peer cloud), and then query their data by outsourcing computations to this network. The secure multi-party computation (or *MPC*) protocols ensure that the servers can never observe any of the plain-text data, and also guarantee that the results they send back are correct. In its essence, MPC ensures that a network of untrusted servers is all that is needed to obtain a secure cloud that preserves privacy.

The literature surrounding MPC is vast, spanning several decades and hundreds of papers, proving what great potential this sub-field of cryptography holds [65]. And yet, practical implementations are few and far between. First and foremost, MPC has poor scaling properties and its performance in the malicious setting is far worse than the semi-honest case, where we assume all nodes follow the protocol.

Second, many impossibility results exist, making the deployment non-trivial. If we are confident that the majority of parties are always honest, then we can ensure privacy, correctness and output delivery, but if this assumption is wrong – security is immediately broken. Conversely, if we would like to protect against a dishonest majority, and specifically – ensure that even if *all* parties are corrupted, privacy and correctness are guaranteed, then we have to settle for a weaker notion of *security with abort*, where even a single party could disrupt the protocol. In other words, it only

takes a single corrupted party to deny service from everyone else (Denial-of-Service attack).

The goal of this thesis is to bridge the gap between the theory of secure computation and its practice. This work focuses not only on implementing a practical system, but also on the applied theoretical aspects that are currently lacking. The ideas brought here are designed to take the first major step towards a scalable MPC platform that can be deployed in practice. This should pave the way for creating a secure cloud platform that could be used to analyze sensitive information.

1.1 Contributions

Secure MPC based on linear secret sharing schemes (LSSS) tends to enjoy significantly better performance compared with other techniques of privacy-preserving computation such as fully homomorphic encryption (FHE). One of the main influencing factors is the information-theoretic nature of homomorphic secret sharing schemes, which do not require expensive public-key cryptography to operate.

However, as it currently stands, MPC has major limitations, which this thesis addresses. It is the main hypothesis of this work that by overcoming these barriers, MPC could eventually take a prolific role in securing systems. This thesis focuses on solving (or significantly improving) four major barriers for scaling MPC:

- **Communication scalability.** As LSSS are only additively homomorphic, each non-linear operation requires a round of communication, which makes the total rounds of communication proportional to the circuit depth. A round requires that all parties synchronize and talk to each other, making scaling the number of nodes in the network intractable. For this reason, all previous work on implementing MPC has focused on the case of 2-3 parties (or a handful of parties at most), which still requires significant amount of trust in each node. As it is reasonable to assume that finding more untrusted parties is easier than mostly-trusted ones, finding a solution that enables scaling the network without adversely affecting performance is crucial. Unfortunately, since we have good

reason to believe that an all-to-all communication is required [53], an alternative approximation is required. Such a solution could be obtained by not having the entire network active in each computation. If done correctly, the security could approximate that of the general case. Several theoretical solutions have emerged in recent years [14] [15], but these were highly in-efficient and their long term security was not analyzed. This work is the first to offer a practical approach, implement it, and offer an (empirical) analysis of long-term security. Long term means that the security applies not only by looking at a single computation, but over a (large) series of computations as well.

- **Active security.** The distributed and untrusted nature of each party makes achieving security against an active adversary more difficult. Actively secure protocols tend to employ more complex building blocks such as zero-knowledge (ZK) proofs and byzantine agreement (BA) protocols. These do not only complicate the protocols and the security analysis, but also significantly impact performance in an adverse manner.

- **Unreasonable assumptions.** In MPC and distributed systems in general, the security model is generally limited to an adversary that corrupts (actively or passively) some portion of the parties. Such classification assumes (almost) nothing else about the adversary in order to derive general results. However, these tend to be weaker in practice, as impossibility results force system designers to assume a global limit on the number of corruptions the system can withstand ($t = \frac{n}{3}$ or $t = \frac{n}{2}$ – i.e., an honest super-majority or majority). The security of MPC (and similarly – of BA) is completely broken if this assumed threshold is not met. A more realistic solution is to model a system that discourages adversarial behavior and provably ensures that the only stable situation is that less than a required threshold of corruption occurs. Other unrealistic assumptions are that of synchronous communication and free reliable broadcast. The first potentially sets an impractically large lower bound on the run-time of even the simplest of computations, while the latter assumes BA is free.

- **Inefficient protocols for integers/reals.** The early days of MPC focused on (im)possibility results in the general sense. With the emergence of Big Data, and given that the seminal results have been well established, in the past decade significant research was dedicated to scalable and practical MPC. This has led to the development of the first MPC protocols over the integers [25] [54] and reals [28] [33]. As it turns out, while simple operations such as addition and multiplication gracefully carry over to integral and fractional domains, most other operations are more complex and therefore less efficient. Even when it comes to a simple protocol of securely comparing two integers, which served as Yao's original motivating example (The Millionaires' Problem), the secure protocol is at least one or two orders of magnitude less efficient than multiplication.

Motivated by these shortcomings of MPC, this thesis presents new tools to overcome scaling and deploying MPC in practice. Perhaps the most novel aspect of this work is the presentation of an MPC model that unifies cryptographic and economic models. While this has been attempted before (e.g., [20]), the results were strictly negative. With the modeling of incentives we are able to show positive results that can also provide a clear rationale to deploying MPC in practice without assuming that some threshold of the parties are incorruptible. A second contribution deals with modeling secure computation together with a blockchain, in order to provide a single more efficient system that could pave the road for scalable MPC. The blockchain assists with defining the economic model, as well as with implementing (nearly) costless BA and decoupling it from MPC. Third, we significantly improve most of the inefficient low-level protocols over the integers and fixed-point numbers. This creates a more optimized framework for MPC over the integers/reals. The main tools to address these barriers that will be developed in this work are described below.

Incentive-Compatible MPC

Incentive-compatibility is a well-studied property of game-theory, formalizing a steady-state of a multi-party system, in which each party only stands to lose by deviating

from the honest protocol. While in cryptography the most basic assumption is that some parties are simply honest (or not), in economics the assumption is that the parties are rational and their utilities can be quantified, such that with a proper incentive mechanism all (or at least most) of them can be discouraged from doing any harm.

Empirically, this assumption seems to hold better in practice than the constrained cryptographic model. Very much like MPC, Byzantine Agreement protocols have traditionally made similar assumptions and were equally absent from practical systems. This has changed with the invention of Bitcoin [1], which is the first seemingly stable large-scale deployment of a BA protocol (or at least some approximate version of it). Despite literally being a decentralized bank that has the ability to mint new currency, and having a collective liquid market-cap worth billions of dollars in the time of writing this thesis, Bitcoin has been highly resilient to attacks and has been able to faithfully maintain a distributed consensus over the correct state of the ledger for over seven years.

Bitcoin is an instance of incentive compatible BA (IC-BA). In practice, some minor attacks have been found to show that this only holds in some weaker form [40] [56], but the general resiliency is so remarkable that many researchers have begun studying this idea [42]. The striking similarities between the security model of MPC and traditional BA has led to the development of a similar framework in this thesis – incentive compatible MPC (IC-MPC). As will be illustrated later, IC-MPC heavily depends on black-box functionalities providing BA over some public state and the ability to transfer payments between parties in a fair and honest manner. Since these are already present in any public blockchain-based system, blockchain technology has an important role in implementing IC-MPC in practice. In that sense, the blockchain plays two roles – it is both the motivation for IC-MPC and also an integral part of it.

A final note on the importance of incentive-compatibility is adoption. In order to encourage growth, a system has to provide some fungible value (and as few barriers as possible) for participants, encouraging them to join. Bitcoin, and this work in the context of MPC, offers such value in the form of monetary incentives. This leads to parties who are a priori interested in this form of return to join the system, enhancing

its security and longevity.

Integrating MPC and Blockchain technology

The usage of blockchain technology in scalable MPC and in privacy-enhancing technologies (PETs) in general extends beyond the economical model of IC-MPC. In MPC in particular, agreeing on some shared public state and verifying who are the honest parties plays a major role in constructing actively secure protocols. The main idea is to use the blockchain as a settlement layer that can be trusted for consensus, which serves as a deterrent that discourages malicious behavior. The blockchain stores commitments to the inputs and can verify transcripts of computations. Parties who deviate can therefore be identified and penalized (or eliminated).

More importantly, the basis of scaling MPC to large networks is based on the notion of quorums. A quorum operates like an elected committee that is trusted with carrying out a computation. Several recent theoretical works have dealt with realizing such protocols [14] [16], but they require an expensive network-wide BA step for agreeing on some public randomness that is used as a seed for the quorum selection protocol. Using the blockchain instead allows us to perform this selection (almost) for free. At the time of writing this thesis, this is the first work to utilize the quorum approach in an efficient manner in order to allow scaling the number of parties in an MPC network.

Optimizing secure computation over integers/fixed-point numbers

Even the most elementary secure computation protocols over the integers and the reals (fixed-point or floating-point representations) are significantly less efficient than addition and multiplication. Secure comparison, which is used as a building block in most of these protocols has been the subject of significant research [55]. And yet, the state-of-the-art currently requires 2 rounds and communication proportional to the bit-length of the inputs. As this is then composed in other protocols, the communication complexity is prohibitive. The base protocol presented in this thesis is optimal, as it requires only 1 round and 1 invocation of the multiplication protocol

for inputs up to 64-bit, which is the most common case. This allows for 1-2 orders of magnitude improvement for comparison directly and more importantly – for all protocols inheriting it. Similarly, a hybrid protocol for larger bit-lengths is presented which significantly improves (almost) arbitrary precision arithmetic.

For many of the common (but inefficient) protocols that are found in any computer program, this thesis shows significant improvements in practice compared to the state of the art. With the exception of secure division/reciprocal and normalization of inputs, for which the first asymptotically sub-linear protocol is presented, all other protocols are focused on practical complexity compared to asymptotic one.

1.2 Structure

The structure of this thesis is comprised of three, mostly independent parts. The common theme across all parts is finding solutions to the problems of scaling MPC described earlier and making the technology more efficient in practice.

- Sections 3-6 describe a generic MPC framework that combines MPC with a blockchain in order to achieve a more efficient system. The last section crosses the border from the domain of cryptography to that of economics, formalizing the IC-MPC model.
- Sections 7-10 go deeper into improving many of the inefficiencies in integer/fixed-point secure protocols, leading to significant improvements over the best currently known protocols.
- Sections 11-12 describe the implementation and an evaluation.

2 Preliminaries

2.1 Secret Sharing

For MPC we are usually interested in linear secret sharing schemes (LSSS), as these are additively homomorphic. A secret-sharing scheme provides two functionalities,

share and *reconstruct* (or *open*). *Share* takes a secret as an input along with the number of shares to split, as well as an optional parameter t in the threshold case (explained in detail below), while *open* takes a list of shares and reconstructs the secret. The underlying goal is to hide a secret by splitting it into shares, such that only a linear combination of $t+1$ of them can reconstruct the secret.

The additively homomorphic property is formally defined as –

$$a + b = Open([a] + [b]), \tag{1}$$

for two secrets $a, b \in \mathbb{Z}_p$, where $[\cdot]$ marks a sharing. In addition –

$$c \cdot a = Open(c \cdot [a]), \tag{2}$$

for a secret $a \in \mathbb{Z}_p$ and a public constant $c \in \mathbb{Z}_p$.

Shamir [48] and Blakley [57] were the first to independently introduce the concept of secret sharing. In the context of a polynomial secret sharing scheme (SSS), we focus on Shamir's scheme below.

Shamir's Secret Sharing

Shamir's secret sharing is a (n, t)-threshold secret sharing scheme, where n is the number of shares derived from the secret, such that a combination of *any* subset $t+1$ shares allows reconstruction of the original data, but any t or less shares reveal no information at all. Shamir's scheme, like many other secret-sharing schemes, is information-theoretically secure.

The scheme works by creating a t-degree polynomial with random coefficients in the field $\mathbb{Z}_p$, where the constant coefficient is set to be the secret. Specifically, to share a secret s, first generate the random polynomial as follows:

$$P_t(x) = s + \sum_{i=1}^{t} r_i x^i, \tag{3}$$

where r_i are random coefficients and by construction $P_t(0) = s$. Then, to create n

shares, simply evaluate the generated polynomial on n different non-zero points (most commonly the series $1, ..., n$ are used). There are several accepted forms of notation for a share – $s_i \equiv P_t(i) \equiv [s_i]$. We will use these different notations interchangeably depending on the context, as each form proves to be more convenient in different cases.

Note that to open a secret, any $t+1$ are enough. Simply use lagrange interpolation over the field to obtain the polynomial $P'_t(x)$ and then evaluate the reconstructed polynomial in $s = P'_t(0)$. If only t or less points are provided then there is at least one or more degrees of freedom, so no information is revealed (even when facing a computationally unbounded adversary).

Additive Secret Sharing

Shamir's secret sharing scheme is sometimes referred to as a threshold scheme or a polynomial sharing. An even simpler, yet related scheme, is that of additive sharing. An additive sharing is strictly an $(n, n-1)$ scheme, as all shares are needed to reconstruct the secret. An additive scheme, as the name implies, splits a secret s into n shares s_i, such that their sum is the secret itself. Namely –

$$s = \sum_{i=1}^{n} s_i \tag{4}$$

To share a secret in this form, the first $n-1$ shares are set to be random values in $\mathbb{Z}_p$ (i.e., $s_i = r_i$) where the last share is set to be:

$$s_n = s - \sum_{i=1}^{n-1} r_i. \tag{5}$$

It is easy to see that Equation 4 holds. Security stems from the fact that the first $n-1$ shares are random values that are not correlated with the secret, and the last share acts like a one-time-pad, giving the scheme its perfect secrecy trait.

One of the benefits of using an additive scheme is that it works well over the ring of integers and not just in a finite field. In this case, the coefficients r_i should be sampled from a large enough domain that is a factor 2^K larger than the domain of the

secret s. This ensures statistical security (compared to perfect), but allows working directly in the ring of integers.

Replicated Secret Sharing

Cramer et al. [38] suggested a threshold secret sharing scheme built from an additive secret sharing scheme. The idea is to split all n parties into all possible maximally unqualified subsets. In other words, given (n, t), we compute all $|A| = \binom{n}{t}$ subsets of size t from the set $\{1, ..., n\}$ and number them from $1, ..., |A|$. We then split the secret to $|A|$ shares ($[s]^{(RSS)} = \{r_j\}_{j=1}^{|A|}$) using the additive scheme from above, and for each party i, we construct the following share vector:

$$[s_i] = \{r_j | \forall j,\ s.t.\ i \in A_j\}. \tag{6}$$

Therefore, each party receives *all* shares belonging to the subsets it is part of. This is why this scheme is replicated, as each r_j is replicated across several parties. By construction, this scheme turns an additive sharing scheme into a threshold one, like Shamir's secret sharing. To reconstruct the secret, any $t + 1$ parties can pool their shares in order to obtain all $|A|$ r_j's, from which they can reconstruct the secret. Any subset smaller than that would be missing at least one share.

Clearly, this variant is more wasteful in terms of space and computation needed, so it is not used in practice directly. Instead, [38] provides an efficient protocol to locally convert a replicated sharing to Shamir's sharing. The reason RSS is so useful is that it allows generating sharings of (pseudo)-random values without interaction, as described below. Together with the local conversion, this implies that we can generate distributed random shares as needed non-interactively.

Pseudo-Random Secret Sharing (PRSS)

Pseudo-random secret sharing (PRSS) is based on the idea that RSS allows us to generate sharings of random and independent values, which are then converted to Shamir's scheme without interaction as well. This provides a distributed random

generator for MPC applications without any interaction.

To achieve this, observe that when performing a RSS, if the secret s is itself a random field element, then all r_j's are independent and random. Therefore, each r_j could be observed as a shared random key for all parties in subset A_j. If the r_j's are added directly, then they conform to a replicated sharing of a single random value. However, if each r_j is used as group key to a PRF $\phi_{r_j}(x)$, then the parties can generate non-interactively any number of random sharings (first RSS and then converting to Shamir). Since a PRF generates pseudo-randomness the scheme is defined as PRSS.

Pseudo-Random Integer Sharing (PRIS).

A variant of RSS called Replicated Integer Secret Sharing (RISS) is given in [58]. Essentially, it is the equivalent of RSS that works in the ring of integers. The randomness is chosen from a larger domain with a security parameter K, thus providing statistical security. Conversion to Shamir is done in the same manner (modulo p). This variant allows generating random integers in a specified domain $[0, 2^k - 1]$.

To sum up, PRSS and PRIS enable generating (pseudo)-random field elements and integers for free (except for computational cost), and are used extensively throughout this thesis. Together with a variant of generating pseudo-random zero sharings (PRZS), they enable an efficient way to generate random triplets offline as in [5].

2.2 Secure Multi-Party Computation

There is an immense body of work around secure multi-party computation. Yao [2] presented the first two-party scheme using garbled circuits, which was later extended to the multi-party case by Goldreich, Micali and Wigderson [9]. An alternative scheme for the arithmetic circuits case, using LSSS was proposed in [6]. A third option that is increasingly becoming more popular is to model secure computation programs using oblivious RAM (ORAM) [7], [8] instead of circuits. ORAM also enables the creation of more efficient oblivious data structures and oblivious sorting algorithms.

Given their information-theoretic nature, as well as their clear fit to the client-server and multi-party settings, MPC using LSSS is generally chosen to implement general-purpose, efficient MPC frameworks. For this reason, this is also the focus of

this thesis. The main idea presented in [6] is that secret data can be shared across n parties using secret sharing. Then, the additively homomorphic nature of LSSS allows computing linear operations (addition, multiplication by a public constant) by asking each party to locally compute these operations on their shares. In addition, [6] provides a secure multiplication protocol that allows the parties to obtain shares of the multiplication of two secrets. Since any circuit could be constructed from addition and multiplication, these protocols are sufficient to show that MPC is complete – namely, every function can be securely evaluated until all parties have shares of the result. At that point, the shares can be combined to reconstruct the result by interpolation, without revealing anything else about the inputs or any intermediate values.

The seminal works around MPC have established feasibility results regarding the allowed number of (passive or active) corruptions. In the perfect and statistical security cases, [6] and [10] proved tight bounds of tolerating at most $t < \frac{n}{2}$ passive corruptions and $t < \frac{n}{3}$ active ones, assuming only secure point-to-point channels. Assuming in addition a broadcast channel, [11] has shown that any functionality can be computed privately and correctly with an honest majority. Similarly, [9] proved similar results of full security against an dishonest minority for the computational security case. If parties cannot abort, the authors have also shown security against any number of corrupted parties.

In the case of a dishonest majority, full security is impossible, as we cannot guarantee output or fairness. Instead, we have to consider the weaker case of security with abort. Recent advancements in MPC against a dishonest majority include [12]. Their scheme incorporates other important improvements to the efficiency of MPC, including the use of randomization triplets [13] that are created in an offline phase, allowing the online (i.e., the actual real-time computation) to run faster. The offline-online model is also known as the preprocessing-model, which has become prolific in any application trying to improve the efficiency of MPC.

In the area of practical implementations, MPC has seen deployments that are limited to a small number of parties, most commonly only protecting against semi-honest corruptions [3], [4]. Active security against an honest minority ($t < \frac{n}{3}$) was

developed in [5]. As most of these systems predate recent improvements in efficiency, they are not geared towards large-scale MPC. Furthermore, these implementations provide little to no support of secure computation over the integers and reals. The only implementations of MPC in these domains are [4] [59].

Security Model

The security model of MPC focuses on the n parties involved in sharing the inputs and then running secure computation over them. The adversary in this model is assumed to be a centralized entity that can corrupt up to t parties. Different variations of this model equip the adversary with different traits. These are usually selected in a way that makes it easier to prove security or to provide a more efficient construct. The goal of this work is to take a different approach – make some assumptions about the real world and protect against the worst-case adversary given these assumptions. It is important to note that while this would seem like a weaker statement, in practice the opposite holds. For example, arbitrarily weakening an adversary (e.g., assuming that it will always follow the protocol, as in the semi-honest case) often results in less realistic assumptions. A summary of the main properties of the adversary are given below:

1. **Passive/Active**. A *passive* (or semi-honest/honest-but-curious) adversary is one that attempts to learn as much information as possible, without ever attempting to break protocol. Therefore, a passive adversary only attempts to compromise *privacy*, but not *correctness*. Conversely, an *active* (or *malicious*) adversary can act in a byzantine manner and attempt to break protocol in addition to attempting to gather information. An active adversary is considered more difficult to protect against, as it could, for an instance, choose to abort, send corrupted version of its shares or even send inconsistent shares to different parties throughout the computation.

2. **Static/Adaptive**. A *static* adversary is one that selects which t parties to corrupt before the protocol starts (and cannot change its mind). An *adaptive*

adversary can corrupt up to t parties at any given time throughout the execution of the protocol. Specifically, the adversary can be selective and corrupt those parties that are more significant in a given execution. This is of particular interest in committee and leader election protocols.

3. **Rushing**. In the synchronous model (see below), a *rushing* adversary is one that can see all messages of the honest parties in a given round, and then decide on the messages the corrupted parties send in that round.

4. **Covert**. More aligned with this work, Aumann et al. [61] presented the idea of *covert* adversaries. These are potentially malicious parties that do not wish to get caught cheating. This definition creates a spectrum between the classic passively/actively corrupt model, and provides security against cheating with some high (but not overwhelming) probability ϵ. The rational behind covert adversaries is that protecting against active adversaries leads to unnecessarily complicated and inefficient protocols, while protocols protecting against semi-honest adversaries are too weak. Instead, covert adversaries tend to emulate some more realistic assumption on real-world adversaries.

5. **Rational**. The covert trait defines a purely cryptographic model and as such, does not tell us anything about what a good choice of ϵ is. Furthermore, like the conventional passive/active definition, it does not capture the idea of *rational* parties, who would attempt to increase their utility by out-smarting the system. The idea of *rational* parties is to connect a cryptographic model to the economic one that tries to model real-world players. In other words, it reasons about *why* parties would act in a certain way, but does not assume that the adversary would automatically fall within a certain classification.

In addition, the adversary may be of bounded or unbounded computational power. In most cases, MPC protocols are proven secure against an unbounded adversary (either with perfect or statistical security), by assuming some ideal functionalities (ideal secure point-to-point channels, broadcast, random oracle model and occasion-

ally CRS). The implementation of these ideal constructs normally leads to systems with computational security.

Proving security of MPC protocols is often achieved using the universal composability (UC) framework [60]. The UC model formalizes a system of parties engaging in interactive protocols (formally, interactive turing-machines (ITM)). Security of a protocol is then proven by showing that these interactive parties can simulate this protocol execution in a way that is indistinguishable (to any observing environment) from a similar ideal functionality executed by a trusted-third party. This notion of security is considered strong, as it ensures that the distributed protocol is at least as secure as the ideal functionality. For this reason, protocols secure in this framework can be composed with other protocols to construct more complex functionalities without compromising security.

Communication Model

Another important aspect of the security of any distributed system is the communication model of the network. The seminal results mentioned earlier are tightly related not only to the type of adversary, but also to the communication model and ideal communication devices that are assumed. Specifically, the completeness results of MPC (e.g., [6]) are defined in a model with assumed secure point-to-point channels and a synchronous model with a global synchronized clock. The latter is a specifically strong assumption, as real-life networks such as the internet are asynchronous in nature. It also makes proving security of MPC protocols easier for the active adversary case, as the adversary has to act in each round or it is easily identified. This ensures that all messages are delivered and an adversary cannot simply abort or delay the protocol.

Work on asynchronous MPC often leads to significantly less efficient protocols and requires making other assumptions, such as having consensus broadcast for free and an eventual delivery of all messages. In this model, there is a tight bound of $t < \frac{n}{3}$ corruptions [64] and we have to settle for the possibility of *input deprivation* – some honest inputs could be ignored. The reason is that if we assume t corruptions, then

an asynchronous protocol should proceed to the next round immediately after $n - t$ inputs are received. Otherwise, if there are t corruptions, the $n - t + 1$-th input may never be received, as the adversary could delay the protocol indefinitely.

Whether the network is synchronous or not, there are two communication methods to consider: point-to-point and broadcast. The latter is commonly used to achieve more efficient secure protocols defined in the pre-processing model [13]. However, this efficiency is only obtained if consensus is assumed to be cost-less (through some ideal broadcast functionality F_{BC}), or at least the broadcast channel is assumed to be reliable (the adversary cannot prevent honest parties from broadcasting messages) and $t < \frac{n}{3}$. In the latter case, secret sharing with error-correction can be used to remove any active faults.

2.3 Blockchain and Cryptocurrencies

In general, a blockchain can be seen as an ideal party (in practice – decentralized) that is trusted with correctness over some public state. This state is stored in the form of blocks, where each block encapsulates all the *valid* transactions that occurred in single round. In that sense, the blockchain is synchronous and the parties maintaining it reach an (approximate) byzantine agreement in each time-step. The time to generate one block is considered the time of the round, and therefore the block number represents a synchronized global clock.

The name blockchain, first described in the Bitcoin whitepaper [1], refers to the fact that there is a single, global list (or *chain*) of blocks. The blockchain is therefore an append only data structure, where all transactions in block $i+1$ occurred after the transactions in block i, but all transactions in a single block are considered simultaneous. In practice, the blockchain provides a form of *eventual consensus*, namely – at any point in time the parties agree on some prefix of the chain, which is immutable with high probability, but the most recent blocks are subject to change due to small forks that occur at the tail. The probability of a fork decreases exponentially (a fork of depth d has probability $O(2^{-d})$) [42]. Correctness is ensured by consensus rules, the most important of which is that the longest valid chain is the correct one. This

ensures the long-term agreement of the state, preventing malicious parties from altering history, as in order to change block at depth d, an adversary would need to spend an effort proportional to that of mining $d+1$ valid blocks, which is considered enormous. Other than the chain, each block's validity is checked by ensuring the block header is valid and that the transactions contained in it are valid. The latter is especially interesting, as validity of transactions is defined by the nodes executing a function (also known as a *script* or a *(smart) contract*) and accepting if it returns true and rejecting otherwise. In other words, the blockchain can be seen as reaching a byzantine agreement over functionalities that can be programmed. With this, we can summarize the main properties and functionalities blockchains expose.

1. **BA and correctness.** As mentioned above, a form of eventual agreement over the state is reached in every round (a single block). The blockchain can be seen as an ideal functionality run by a trusted-third-party (TTP) that all parties have black-box access to. This functionality also provides consensus broadcast for free, although one that might be slow, so in reality it is only used as a last resort (we refer back to this in the implementation).

2. **No privacy.** Since all information on the blockchain is public (this is needed to reach agreement), we conclude that it provides no privacy guarantees over the inputs.

3. **Incentives.** In public blockchains such as Bitcoin, incentives are key to ensuring nodes' honesty. This is the form of IC-BA alluded to earlier, and also the motivation for extending this idea to MPC in this work. As such, a blockchain provides us with an additional ideal functionality for transferring value between parties, including locking a deposit on-chain.

4. **Global synchronized clock.** Since in every round the nodes reach consensus over a new block, this implies that they also reach an agreement on a global (discrete) time.

It is important to note that these properties are exposed through ideal functionalities the parties in our MPC network use. In other words, when we formalize our mixed MPC-blockchain model, we treat the latter as a centralized ideal party. In practice, the MPC nodes may also be blockchain nodes, but this does not affect the model.

Notation. Formally, we will use the model of Kosba et al. [43], which provides us with wrapper functionalities for contracts and protocols. A contract is simply a function that is run by the blockchain, whereas a protocol is run locally by each of the parties. Parties' use protocols to interact with contracts on the blockchain and vice versa. It is interesting to note that their framework is UC-compatible, which is important for composing UC-secure MPC protocols.

We will also use the same notation. T marks the current round (i.e., global time); ledger[p] the current account balance of p; and access[o] the list of services owner o has approved. We will use x_{ref} for the on-chain object referencing a secret that is secret shared among computing parties in P. These will include the current quorum, when the secret was last updated, and history of commitments – starting from the dealer. Essentially, the blockchain holds all public information that it needs to use in order to maintain the system and identify cheaters.

2.4 Game Theory

Game theory explores the science behind the interaction of rational agents trying to maximize their utility (i.e., their expected return). This is in contrast to cryptography and distributed systems which put stringent (but fixed) constraints on the parties. Cryptography has the advantage that no assumptions on the preference (utility) of the agents is assumed, but at the cost of general assumptions about the adversary which may not hold in practice. Game theory on the other hand assumes something about the parties' preferences, but does not make any additional assumptions about their behavior; that is determined mathematically by the model, where the nodes (may) converge to some equilibrium in which no party can gain anything by deviating. In that sense, rational parties are harder to handle, as they do not follow the universal

assumptions set forth by the cryptographic model. If a system is poorly designed, the equilibrium might lead to a bad social outcome.

One simple example could be of two criminals accused of a crime. If even one of them pleads guilty, both criminals would end up in prison for a long period of time. Alternatively, if both claim they are innocent they walk free. This is a modified version of the *prisoner's dilemma* in which there is no incentive for either party to turn on each-other, so both parties would end with the best outcome for them (no punishment). However, from a social perspective, the outcome is bad and the system is clearly broken. It is fairly easy to design a game-theoretical mechanism that protects against that. However, if we force a cryptographic model that assumes that at least one of the two criminals is always honest, then that would appear secure in theory where in practice, this assumption would not hold, as parties are rational in reality.

In terms of notation and tools needed, we use the standard definition of finite extensive form games, as well as a mediator as a communication device (see [49] for details). Essentially, players do not interact directly with each other, but rather they simultaneously send a message to the mediator in every round, who in turn computes a (possibly probabilistic) function and sends the output back. Formally, in every round t of the game Γ, player i sends input I_i^t to the mediator m, who computes $P^t : I_1^t \times ... \times I_n^t \times r \to O_1^t \times ... \times O_n^t$ over all the inputs in this round and optionally some random bits r, then sends each player i its output O_i^t. Note that we will assume that the full output O^t is always common knowledge.

As usual, we use σ_i to denote the strategy of player i, σ_{-i} to define the strategy of all players other than i, and $\vec{\sigma} := (\sigma_i, \sigma_{-i})$ to denote the strategy profile of all players. Let $u_i(\vec{\sigma})$ mark i's utility if the strategy profile $\vec{\sigma}$ is played. We say that σ_i is a dominant strategy for player i if for all $\sigma_i \neq \sigma_{i'}$ and for every σ_{-i}, we have: $u_i(\sigma_i, \sigma_{-i}) > u_i(\sigma_{i'}, \sigma_{-i})$. Intuitively, this implies that a player has unique best strategy that she will always play, regardless of how other players behave. If all players have a strictly dominant strategy, then it means that they will always play it in every run of the game. We call this a (strictly) *dominant strategy equilibrium*, which is

the strongest type of equilibrium a game can have. A mechanism is informally the design of a system that attempts to achieve some social objective through the use of incentives. In our case, this is an MPC network and the goal is to maintain privacy of the inputs and correctness of the outputs. If a mechanism is designed in a way that being truthful (i.e., acting honestly) is the dominant strategy equilibrium, we say that it is *incentive compatible*.

2.5 Security Model of This Work

As hypothesized in the introduction, one of the major barriers for deploying MPC solutions is the unrealistic security model. It is in some senses too strong and in others too weak. These lead to protocols that will either not be secure in the real-world, or will be too impractical to deploy. Based on the definitions above, we make the following (qualitative) axioms:

1. *The passive/active adversary model is insufficient.* Passive security assumes that no system in the network is operated by a malicious party and that no system can be hacked or even encounter a power-failure. Conversely, active security requires limiting the number of failures, requiring a super-majority of honest nodes, or otherwise it leads to too inefficient protocols in practice. In the case of dishonest majority [12], a single party can put the entire network in a state of a dead-lock.

2. *Byzantine Agreement is not free.* Assuming that a reliable broadcast is given is a strong assumption. In practice, a corrupted party may send different messages to different nodes, so a byzantine agreement protocol should be implemented to reach consensus on *every* message that is broadcast in the network. Even for a solution with authenticated channels using PKI (i.e., a computationally secure solution), protecting against $t < n$ corruptions requires $t + 1$ rounds, even if there are no corruptions in practice at all. This by itself is too expensive to allow for any real-life computation to run in practice. Furthermore, corrupted

parties may try to eclipse honest parties, preventing them from being able to send messages using the broadcast channel.

3. *Synchronous MPC imposes major delays.* Since most MPC protocols are described under a synchronous network assumption with a global shared and synchronized clock and an upper bound on the time of each round, this implies that the constant time of the round would potentially dominate the entire computation. Assume for example that the time-per-round is set to 10 minutes. This is done in order to allow honest parties who are delayed by network problems (or denial-of-service attacks) to resume operation. In reality, even 10 minutes is not likely to be sufficient, but for this example we assume it is all that is required. Note that evaluating a circuit C (optimized for low-depth of multiplication gates), would still require at least $10 \times depth(C)$ minutes to execute. Alternatively, using asynchronous MPC requires other strong assumptions and inefficiencies.

The main idea brought in this thesis is to find computational solutions to these ideal functionalities that are not free in practice (BA and synchronicity) and make more realistic assumptions on the adversary. Generally-speaking, we prefer to allow more corruptions (including a dishonest majority), but we assume that parties care about incentives, and that being caught incurs a penalty. With these assumptions, we are able to provide efficient results *in practice*, and also protect against situations such as *aborting* and *input deprivation* that prevents *liveness*.

The two constructs that provide these abilities are rationality and the blockchain. In terms of security, it implies that the security assumptions of the system are reliant on the computational security of the blockchain. In that sense we take the approach of [43] and assume that the hosting blockchain is secure, and that it provides ideal functionalities such as a global synchronized clock and incentives transfer. The blockchain itself is a synchronized construct with a large round-time (an orders of seconds or minutes, depending on the actual implementation). However, by decoupling the blockchain from the MPC parties, the underlying MPC protocols can

run asynchronously without incurring the penalty of a synchronized system. We solve the input deprivation problem by assuming that inputs are secret-shared a priori and then live in the network (with a mobile proactive secret sharing scheme (MPSS) [22]). Furthermore, we use the blockchain to provide a broadcast with guaranteed eventual delivery (this is true to some close approximation [56] [40]). Parties can then execute secure computations in the network in a fast, asynchronous manner, while having the blockchain provide synchronization and time-outs on the rounds as a form of additional security. This enables us to obtain the best of both models, given assumptions on the security on the blockchain.

In addition to the blockchain, another less conventional assumption is that of rationality. We later define and prove the IC-MPC model and show that rational parties would not be actively malicious if they can be penalized, while honest parties are rewarded. We do not require all parties to be rational, but if that is the case then the system would converge to the semi-honest model.

To summarize, our model is comprised of an asynchronous network with a global synced clock, with an upper bound on the time-per-round, as well as a reliable broadcast with eventual delivery and consensus. We also work with mostly rational parties with a utility function that discourages them from being caught cheating. Furthermore, it is assumed that they are incentivized when they follow the protocol, so delaying execution decreases their utility, thus normally the network would work in an asynchronous manner and not wait for the actual round to complete.

The security model of rational parties that are incentivized introduces a new school of thought as we explain in Section 6. In a nutshell, in almost all previous work on MPC the assumption on the adversary is that if it is actively corrupted (correctness), it also attempts to learn sensitive information (privacy). In general, treating active corruptions is more difficult and requires to reduce the threshold t. In our case, rationality ensures that active corruptions are sparse (since they are detectable), but cannot assume that rationality can prevent honest-but-curious parties to collude (as this is undetectable). So in contrast to most previous research, the security model of this thesis discourages the adversary from being active but not passive. Therefore, it

is reasonable to assume that if the adversary is rational (for the most part), then in practice it is more passively than actively corrupt.

3 Overview and Design

In this section, we give high-level overview of the MPC framework. This is a functional explanation that describes the different actors and components. These are later formalized into a series of generic protocols that define how these come together into a single system (Section 4) and finally implemented (Section 11) with all the additional optimizations done on MPC in Sections 7-10.

At a high level, our framework serves as a distributed cloud system that ensures both privacy and integrity of the data it holds. The system also allows any type of computation to be outsourced to the cloud while preserving the privacy of underlying data and correctness of the result of the computation. A core feature in the system is that it allows the owners of the data to define who can query it. This ensures that the owners themselves control who can query their data, in which case the approved parties, which we call *services*, only learn the output, without ever seeing the raw data. Since all computations are done using secure MPC protocols, no other party learns anything else.

There are three types of entities in the system where in practice, each entity can have multiple roles. *Owners* are those sharing their data into the cloud and the ones that control who can query it; *services*, if approved, can query the data without learning anything other than the result ; and *parties* (or *computing parties*) are those nodes who provide computational and storage resources in exchange for monetary rewards. Note that traditionally, owners are defined as *input parties* and *services* as *output parties* (collectively – the clients), while the computing parties are sometimes denoted as the *servers*. Our definitions simply provide contextual meaning.2

Conceptually, owners and services are users of the system who see the parties as a remote cloud they interact with. Parties are the nodes that actually construct the cloud and can be likened to physical servers. However, unlike a normal cloud where

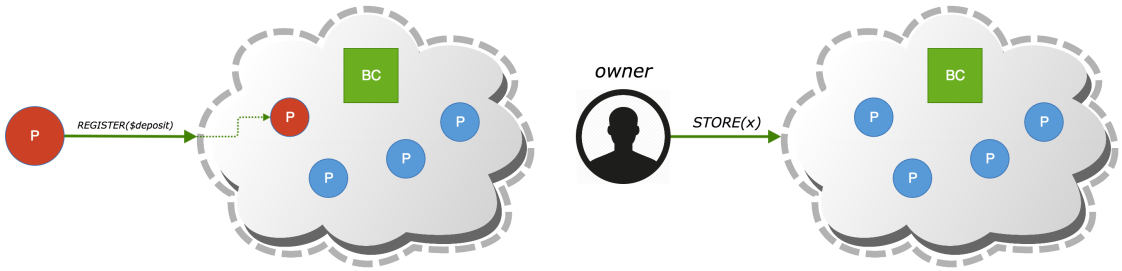
the servers are centrally owned, the system is decentralized. Another way to think of computing parties is as general-purpose miners (as in Bitcoin), who are willing to do computational work in return for rewards. An additional special (ideal) party is the blockchain. The blockchain is also part of the internal cloud and has a key role in it. While in practice, the blockchain is a decentralized party of its own, it is convenient to formally model it as a single (ideal) party that provides the following properties: *correctness*, *incentives* and *synchronized global clock*. This comes at the cost of leaking all information publicly (i.e., no privacy).

To illustrate our system at a high level, Figure 1a shows what interacting with the system looks like for each of the three types of entities. In the first sub-figure, computing parties register on the blockchain and connect to other registered parties to form the distributed cloud. Each party is required to have sufficient funds that the blockchain can lock away in a security deposit account, to be used in the event of dishonest behavior. In general, the blockchain is entrusted with doing all the book keeping, as well as identifying cheaters and handling disputes (i.e., functionalities that relate to achieving consensus over a public state). In addition, it is trusted to act according to the protocol and facilitate transactions that pay (or penalize) for good (or bad) behavior.

Once the network is formed, owners can share data into the system. While the underlying process, which we will describe shortly, is quite different than traditional systems, this is easily abstracted away by a software library that runs on the client (owner or service). From the owner's perspective, sharing data requires issuing a single *store* call with the data to the network, as seen in Figure 1b. No further interaction from the owner is needed. Also in the purview of the owner is informing the parties who can query (i.e., compute over) the data (see Figure 1c), in a privacy-preserving way. The owner can always change the set of permissions, adding new services or revoking access from previously approved ones.

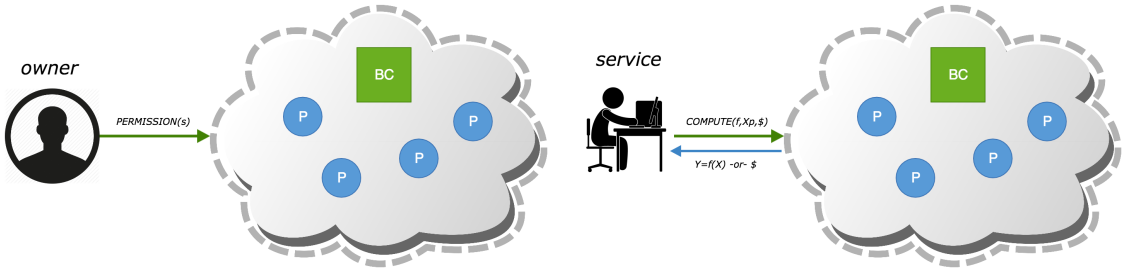
Finally, most of the action occurs in the computation itself. As seen in Figure 1d, a service sends a computation (i.e., a program) to be evaluated in the same way it would do with a normal cloud. The computation itself includes payment for the resources

parties provide. Subject to the model assumptions, the service is guaranteed to either get the *correct* output, or to have its payment refunded. Similarly, the owners are guaranteed privacy.



(a) New party registers to the cloud; Deposits money.

(b) An owner secret-shares data to the cloud.



(c) An owner approves a service to query her data.

(d) A service sends code to be computed securely. Only obtains the result.

Figure 1: High-level overview of how different players interact with the cloud.

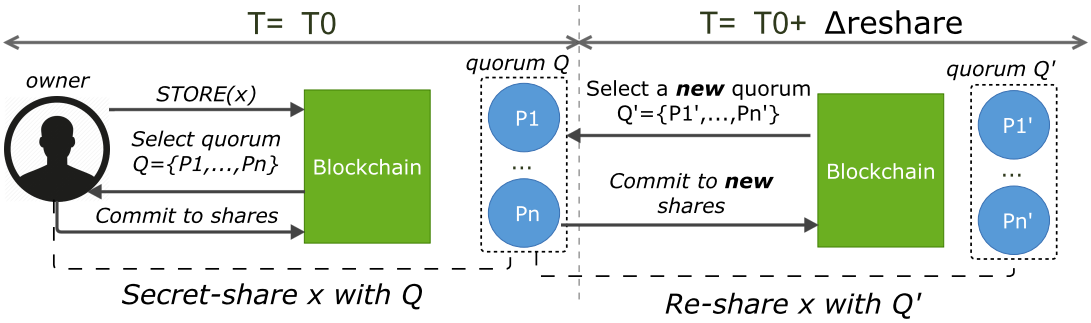


Figure 2: A look inside the cloud-storage. What happens after an owner secret shares data.

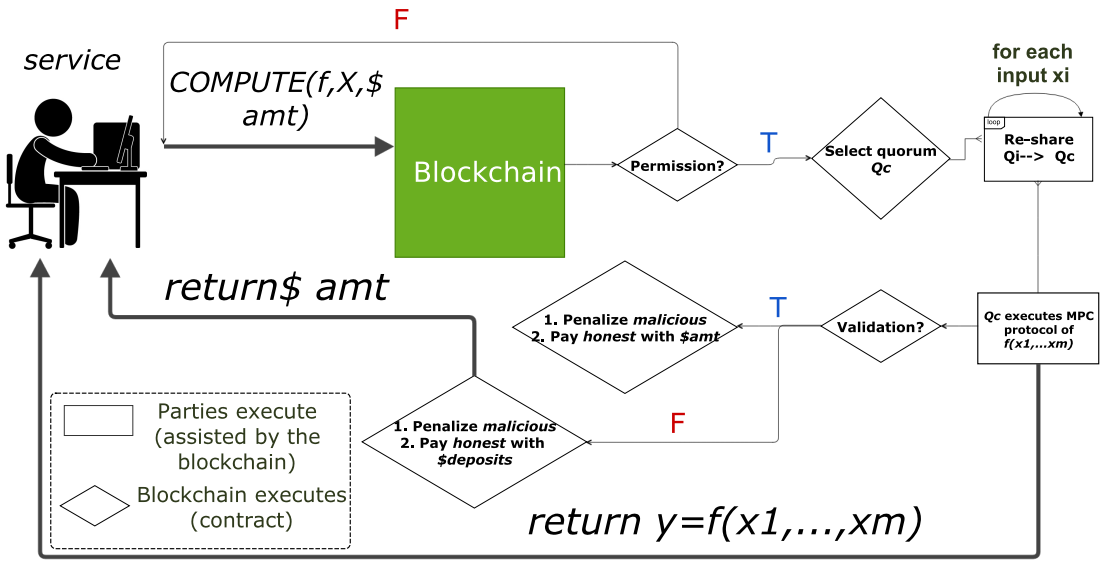


Figure 3: A look inside a computation. What happens when a service sends code to the cloud.

3.1 Under the Hood

So far we have described the different players in the system and have shown how from their perspective, the system appears like a standard cloud, but with better security and privacy properties. An overview of how the network internally provides these guarantees is given, with the focus on *storage* and *computation*.

First, we assume that owners, services and the computing parties themselves are all connected to a blockchain and use its broadcast channel. To simulate pair-wise secure communication, a party sends a message encrypted with the corresponding public encryption key of the other party. By design, the blockchain itself is a network with broadcast that has strong eventual delivery guarantees. Therefore, when we formalize our protocols, we will treat every party that is interacting with a contract on the blockchain as if it broadcasted a message with guaranteed delivery. Note that this provides consensus broadcast in a synchronous communication model – messages cannot be delayed for more than a round (in practice, for a constant number of rounds, until consensus is guaranteed with very high probability). The extension to the asynchronous case will be argued through rationality. The network has an (unreliable, or weakly reliable) broadcast channel that is asynchronous (which is free

as it requires no consensus). Theoretically, an adversary can delay messages, either its own or even those of honest parties (by attacking the unreliable broadcast channel), until the end of the current round. However, as is later explained, a rational adversary could only decrease its utility by delaying the protocol, as that would require more time spent on the current computation, whereas the reward per computation is fixed.

Storage. Figure 2 shows the low-level process that is invoked when an owner shares data. The owner starts by sending a store request to the blockchain. The blockchain invokes a contract (i.e., a function) that selects a set of registered parties at random to store the data, from hereon referred to as a quorum (or a storage quorum), and sends back to the owner this set. The owner then verifiably secret shares the data locally, and encrypts each share with the corresponding party's public encryption key (these are kept on-chain). The set of encrypted shares and their commitments are sent to the blockchain. Once in every some time period (potentially variable and random), the blockchain will trigger a re-sharing protocol. The protocol is an adaptation of the *mobile proactive secret-sharing scheme* (MPSS) described in [22] and explained in Section 5. Essentially, in the re-sharing protocol, the blockchain selects a replacement quorum to store the data. Then, through a secure protocol, the shares are transferred from the old quorum to the new. If any share has been lost in this time period, the parties in the old quorum can collaborate to securely restore it. This process is periodically done for every piece of data shared into the system, and ensures that data never stays too long in a single quorum, which could be slowly corrupted. Recovering lost shares also ensures the integrity of the data.

Computation. To request a computation, the service begins by sending a compute request to the blockchain, providing the function, a reference to the data, which could originate from multiple owners, and a payment. As illustrated in Figure 3, the blockchain executes a contract that verifies that all owners of the data have approved this service. In practice, the parties themselves execute this process off-chain asynchronously, and the blockchain only executes this at some point in the future (e.g., in the next round), in order to determine whether a request was valid. Parties will not honor the request if it is an invalid one, for the same reason the blockchain would

not, as that implies the payment to them would not be accepted as well. Our scheme is based on [24] and improves on it, as we describe later. Once the service is verified, the contract selects a one-time computation quorum at random and asks all storage quorums holding the relevant data for this computation to do a re-sharing to the new computation quorum (in parallel), namely – supply the new quorum with the secret shared inputs for the computation.

Once the computation quorum has all the inputs, it engages in a secure multiparty computation of the function. Then, there are two possible outcomes. Either the computation succeeds, in which case the service can reconstruct the result and its payment is split between the honest computing parties; or, the computation fails and the payment is refunded to the service. In any case, the blockchain will identify cheaters (after the fact) who will not be paid, but instead – penalized. Honest parties are always paid, either by the service or by the corrupt parties.

The overall computation/storage process can be seen as occurring in three distinct parts:

1. **Pre-processing.** Both sharing of the inputs and pre-processing of data-independent randomness happen in this step. This is the only step the owner is involved in (non-interactive later on).

2. **Online phase.** The actual online MPC is invoked. The overhead is that of concurrent re-sharing. This is negligible in the depth of the circuit to be computed.

3. **Post-processing.** The blockchain reaches consensus on the identity of honest/malicious parties (including the service) and splits the payment/penalties accordingly.

The first two phases (offline and online) comprise the standard pre-processing model of MPC. The only difference is that the sharing of inputs has been pushed to the offline phase (since that has to occur synchronously). Given that, input deprivation is not a concern and the online phase can occur asynchronously. The post-processing

step also occurs synchronously by the blockchain, but it is decoupled from the online phase. This of course implies that correctness of the output could be overturned sometime in the future, but due to the dominant strategy equilibrium, this is unlikely to occur frequently (or at all).

4 Generic Protocols

The protocols in this section define the overall framework. They formalize the details of the previous section. Using the same terminology of [43], protocols that are executed by the blockchain are referred to as *contracts*, whereas a protocol executed locally by each party is called a *protocol*. The description in this section follows the synchronous model for simplicity. The extension to the asynchronous case through rationality is explained in the next section.

4.1 Interface

The interface contract described in Contract 1, serves as an entry point to the system. It exposes the main functionalities illustrated in section 3: *register* for the computing parties providing resources to the network; *share* and *access*, allowing owners to store data and set permissions; and *compute*, allowing services to send arbitrary functions to be evaluated securely (through MPC protocols) on data that lives in the network, provided they have the appropriate permissions. In the rest of this section, we will model the different components of the framework in detail and explain how they connect.

4.2 Registering

Only parties that have at least $\$deposit_{TOTAL}$ in their account can register to become a computing party, in which case this amount is held by the contract. This becomes the *security deposit account* used by the contract to penalize a party for cheating and

Contract 1: InterfaceContract

Init. Set $access, P := \{\}$, global timing and payment parameters (ΔT_{round}, $\Delta T_{dispute}$, ..., $\$deposit_{TOTAL}$, $\$deposit$, $\$amount$)
Register. Upon receiving (*register*) from some party p_i:

- assert $ledger[p_i] > \$deposit_{TOTAL}$
- if $p_i \notin P$, add p_i to P and lock $\$deposit_{TOTAL}$ amount in $ledger[p_i]$

Share. Upon receiving (*share*, $\vec{COMM}_x$, $\{COMM^{(0)}_{[x]_i}\}_i$) from some owner o:

- Select a random quorum $Q \subset P$; store an x_{ref} object containing meta information (o, Q, x's commitment, and current time)

Access. Upon receiving (*access*, $\{s_i\}_i$) from some owner o, set access[o] := $\{s_i\}_i$.
Compute. Upon receiving (*compute*, $f, \vec{x_{ref}}$) from some service s:

- Assert that s can query *all* $\vec{x_{ref}}$.
- Select a random computation quorum $Q \subset P$. Then, for each input reference $x_{ref,i} \in \vec{x_{ref}}$, call Re-Share($x_{ref,i}.Q \to Q$)
- After all inputs were transferred, instruct each $p_i \in Q$ to initialize ComputeProtocol(f, $x_{ref,i} \in \vec{x_{ref}}$, s).

Timer. For each x_{ref} stored in the system, if the secret requires updating, then:

- Select a new random quorum $Q \subset P$.
- Call Re-Share($x_{ref}.Q \to Q$)
- Instruct $x_{ref}.Q$ to delete the underlying secret and set $x_{ref}.Q := Q$.

will be a key component in the security and efficiency of this system. Note that a deposit also provides some protection against sybil attacks, since participating has an associated cost.

For practical considerations, the sum $\$deposit_{TOTAL}$ serves as an upper bound that is higher than the expected penalty for cheating in a single protocol execution. This prevents the need for parties to constantly refill their deposit account whenever they pay a penalty. Only when the account drops below a certain threshold, then will the contract decide to unregister the party. For brevity, we do not include the unregister activation in the interface contract, and only mention that it can be triggered either by the contract or by the party itself. Before releasing the remaining funds in the security deposit account of a party, the contract will initiate an ordered removal protocol which ensures a party is not presently participating in any executions and that all of its shares are transferred to a random replacement.

4.3 Access Control on the Blockchain

In a work published by the author of this thesis [24], it is described how a blockchain could be used to store and enforce access-policies to data that are stored off-chain. The definition of owners and services matches the one presented earlier. The protocol starts with an owner sending a signed transaction to the blockchain with a list of public keys of services she wishes to approve. Then, when a service asks an off-chain storage for this owner's data, the storage server, be it centralized or distributed, can check if it has permission on the blockchain. In short, the scheme uses the integrity of the blockchain to log access-control policies.

The following is an extension of this idea. At any time, an owner can change the list of approved services on the blockchain, as is formalized in the *access* call of the interface contract. Unlike the cited paper, only the owner can ask for the raw data back, so we do not trust the service with the data. An *approved* service can only send a function to be evaluated securely, and obtain its output. Also, we do not need to trust an off-chain storage. Much like Bitcoin, our network is decentralized and incentivized, so parties will only get paid for doing work that is according to

the protocol. If a service is not approved to query an owner's data, then the parties should drop the request.

4.4 Quorum selection

Earlier it was illustrated how quorums play a key component in the system. Each time a new secret is shared, a storage quorum is selected to hold it and this quorum is refreshed every period. Similarly, every time a computation is requested, a new computation quorum is selected to execute it. The rationale behind quorums is to allow scaling MPC to large networks, since non-linear operations require an all-to-all communication between parties ($O(n^2)$). In contrast to all previous MPC systems, in our design, fast quorum selection implies that scaling the network could increase (instead of decrease) performance, by leveraging parallelism and load-balancing techniques. The idea of quorums has been heavily researched in recent years (e.g., [16] and [14]). The cited papers rely on [50] BA protocol to reach consensus on a valid set of quorums. This protocol is expensive and it cannot be efficiently used in a highly dynamic network like ours. In practice, the theoretical solution is only practical if the BA protocol's cost can be amortized across many computations. However, since in the real-world the adversary is adaptive and this protocol only protects against a static adversary, quorums should be selected frequently, making this solution impractical. It also protects against $t < \frac{n}{3} - \epsilon$ faults, but given an (eventual) consensus functionality provided by the blockchain, we do not incur a similar constraint.

The solution proposed in this work is to replace the (relatively) expensive BA protocol, which is used in order to agree on a seed for selecting random quorums, with one that uses the blockchain directly as a source of agreed-upon public randomness (beacon) for selecting quorums. This has been previously and concurrently explored in [70] [69] (for other applications). Now we can achieve consensus everywhere on the identity of the quorums without cost. Furthermore, our quorum size is independent of n, achieving better overall scalability and flexibility.

Note that in practice, we do not select a single quorum for storage or computation. Instead, a quorum is selected for each share, namely – we use 2-level secret sharing to

better protect the privacy of the data. The intuition is that we need to ensure with high probability that a bad quorum will not be selected. This will be revisited and analyzed in Section 12. However, for the sake of simplicity describing the protocols assuming a single (flat) quorum would suffice.

4.5 MPC Protocol

This section describes the generic protocol for MPC computation in a setting where the parties can access a blockchain and are incentivized to act honestly. In later sections many of the particularly important MPC elementary protocols will be optimized, in order to make any complex function (which is a composition of these building blocks) run faster. For now the focus is on evaluating (any) generic function in a MPC, so the focus is on a circuit with addition, multiplication and output gates only. The given protocol achieves efficiency that is comparable to a semi-honest execution (for any function), assuming a blockchain and the IC-MPC model that assumes rationality. In other words, evaluating a function securely is expected to run as fast as the optimistic case where the adversary is passive, without explicitly assuming it. This also provides guaranteed output delivery against a dishonest majority, assuming enough parties that the adversary controls are rational.

The key is to design an optimistic protocol that runs (almost) like a semi-honest execution in the online phase, without resorting to expensive validations. Instead, parties collaboratively construct a transcript on the blockchain as computation progresses. The goal is to provide the blockchain with enough information to identify cheaters even if the computation fails (i.e., too many corrupted parties). In fact, even if all parties are corrupt, the blockchain can detect it. As we will show when we formalize IC-MPC, an external mediator, which we call an *observer*, who can detect cheating regardless of the number of corruptions, is the key property for ensuring that following the protocol is the dominant strategy and that it is incentive-compatible. This observer is a formalization of the blockchain in the game-theoretic sense, which can detect and subsequently reward/penalize parties.

As mentioned earlier, the verification work is done *eventually* in a post-processing

phase, since in practice the parties are allowed to interact off-chain through asynchronous communication without waiting to synchronize with the blockchain. This is significantly faster than protocols that rely on expensive cryptographic primitives such as zero-knowledge proofs (e.g, [6]), or protocols that could run indefinitely in the presence of malicious nodes (e.g, [12]), since malicious nodes are never detected and cannot be removed.

The full details of our protocol are given in the on-chain Contract 2 and the local Protocol 3, so we only provide an intuitive description here of its workings. We have already described how a computation request initiated by a service is routed to a new computation quorum who holds shares of all inputs. The analysis here focuses on the part that follows, namely – the MPC computation itself, which starts by the parties initializing Protocol 3 with the function, their shares, and the requesting service.

The computation then proceeds in rounds. The parties execute the computation locally, which they can do until a multiplication (or output) gate is reached. If it is a multiplication gate, then they randomize their shares as in [13] and send it to the blockchain. When sufficient parties have provided their masked shares, the parties call *sync* again to proceed with the computation. This process repeats until an output gate is reached. In which case, the parties use the service's public encryption key to encrypt the share of the randomness masking the output, and broadcast it alongside the masked shares. This ends a normal computation and is done asynchronously from the blockchain. If for some reason the computation failed (i.e., there were too many corrupted shares in some round), then the parties simply abort. The blockchain will figure out after the fact who cheated and who was honest.

Similarly, from the blockchain's perspective, each round has a time limit, after which *endround* is called, where the blockchain runs a quick verification to see if the computation can proceed or not (adding cheaters to a corrupted parties list along the way). Depending on the secret-sharing threshold, a quick verification could be

Contract 2: ComputeContract(Q, f, s)

Init.

- set state := NORMAL, $corrupt, transcript := \{\}$, the time lock parameters and payment constants $\Delta T_{round}, \Delta T_{dispute}, \$amount, \$deposit$, and set cround := 1, $T_{round} := T + \Delta T_{round}$.

Sync. Upon receiving (*sync*, round, v_i, e_i) from $p_i \in Q$

- if $T < T_{round}$, set $transcript_{round}[p_i] := v_i$ (if it does not exist).
- store e_i if $e_i \neq \perp$ and p_i has not shared output yet.

EndRound. Called on input (*endround*) from the contract.

- add all missing inputs to the set *corrupt*.
- reconstruct $transcript_{round}$. If succeeded:
  - add any parties who aborted or (clearly) sent bad shares to *corrupt*.
  - if cround = last round → call (*finish*, *true*)
- else (computation ended early) → call (*finish*, *false*).
- cround := cround + 1; $T_{round} := T + \Delta T_{round}$.

Finish. Called on input (*finish*, result) from the contract.

- if result = true
  - select $p_{test} \in Q \backslash$corrupt at random
  - result ← call (*verify*, p_{test})
  - if result = true // i.e., still checks out
    - Set state := DISPUTE, $T_{dispute} = T + \Delta T_{dispute}$ and return.
  - else // all seemingly honest parties cheated
    - remove from *corrupt* all existing parties except aborters
    - add to *corrupt* all previously considered honest players
    - call (*pay*)
- else // computation failed
  - for every party $\notin$ *corrupt*, call (*verify*, p_i)
  - call (*pay*)

Verify. Called on input (*verify*, p_i) from the contract.

- execute f using the computation transcript and commitments to p_i's shares of the inputs.
- let sim_r be the result of simulating round r on behalf of p_i.

Pay. Called on input (*pay*) from the contract.

- if $|Q| - |corrupt| > m$ // m is the SS threshold.
  - $ledger[s] := ledger[s] - |Q| \times \$amount$;
  - $ledger[p_i] := ledger[p_i] + \$amount$ for every honest p_i paid from s
  - $ledger[p_j] := ledger[p_j] - \$deposit$ for every corrupt p_j.
- else // computation failed
  - $ledger[p_j] := ledger[p_j] - \$deposit$ for every corrupt p_j
  - $ledger[p_i] := ledger[p_i] + \$amount$ for every honest p_i paid from the corrupted parties
- Set state := COMPLETE

Timer.

- if state := NORMAL and $T > T_{round} \rightarrow$ call (*endround*).
- if state := DISPUTE and $T > T_{dispute} \rightarrow$ call (*pay*).

attained based on the threshold value of the SSS. For $t < \frac{n}{3}$, simple error correction is sufficient, and for $t < \frac{n}{2}$, a more clever scheme with larger shares can be used [62] [63]. For the general $t < n$ (which is of interest in this work), only certain faults like aborting can be identified between rounds, and the actual validation is checked after the computation has completed. This occurs in the last round, or if the computation failed prematurely. The contract moves to a post-computation procedure of verifying the computation and classifying all nodes as either honest or corrupt (see *finish* activation). In reality, the parties themselves operate asynchronously from the blockchain, so after every MPC round they should perform some quick verification (that is cost-less and non-interactive). This can be done with error-correction as above.

The final verification works by simulating an execution of a single party from start to end, using commitments to the inputs and the transcript. This, in combination with reconstructing the masked polynomials the parties share in each round, would reveal all the corrupted parties. The only private information left at the end of a

Protocol 3: ComputeProtocol

ComputeProtocol($f, \vec{x_i}, s$) for a party $p_i \in Q$:

- **Init.** Maintain a state for the computation, i.e., Let $round := 1, f_{ptr} := f, y_{curr} := \vec{x_i}$.
- **Sync.** On input (*sync*)
  - if $round > 1 \rightarrow$ query the contract's transcript and update the interim y_{curr} values.
  - execute $f_{ptr}(y_{curr})$ locally until interaction is required (f_{ptr} points to current op.)
  - let v_{curr} be the masked shares for this gate.
  - set $e_{curr} := ENC(r_i, s.epk)$ if this is the output gate, or $e_{curr} = \perp$ otherwise.
  - send (*sync*, round, v_{curr}, e_{curr}) to the contract
  - $round := round + 1$
- **Discard.** On input (*discard*)
  - assert state := COMPLETE on the blockchain
  - erase $\vec{x_i}$ and interim y_{curr}.

ComputeProtocol for the service s:

- **Compute.** Send input (*compute*, f, $\vec{x_{ref}} = \{x_{ref,j}\}_{j=0}^{m}$) to the *InterfaceContract*.
- **Result.** On input (*result*)
  - for each p_i
    - query v_i, e_i from the contract;
    - $r_i := DEC(e_i, s.esk)$
  - reconstruct v, r; $y := v - r$ is the output
  - (Optional) if state := DISPUTE and $T < T_{dispute}$:
    - let accused := list of parties who did not encrypt correct shares.
    - send (*dispute*, accused, $s.esk$) to the contract.

seemingly successful execution are the encrypted shares of the masks of the output. It may be the case that parties did not encrypt the correct share. For this reason, the contract allows the service to dispute before the blockchain marks this computation as successful. After the dispute period has passed, the blockchain settles the score with all parties (see *pay* activation). If the computation succeeded, honest parties are paid using the payment the service provided. Corrupted parties are not paid *and* penalized (their funds are burned). If the computation fails, the service does not have to pay and the penalties are used to pay the honest parties.

Note that for all parties, the outcome is binary: honest behavior is paid the same amount, while bad behavior is penalized. For the service, an output requires payment, but a failed computation does not. This is crucial, since otherwise some parties may employ malicious strategies to maximize their profits. For example, if an honest party was able to both collect the payment and a penalty, then it may attempt to attack its peer directly so that the blockchain would believe it cheated, thus paying the supposedly honest node twice. In addition, the reason why corrupt nodes are penalized, instead of just not receiving payment, is to have an alternative source of income for the honest nodes in case the computation fails[1].

Finally, at the end of the computation, all honest parties call their local *discard* activation and delete their shares of the inputs as well as other interim values. Since before a computation starts, the inputs are re-shared to the selected computing quorum using a new one time sharing, there is no point, even for the adversary, to continue holding them.

4.6 Security Analysis

Intuitively, the framework preserves (computational) privacy of the inputs regardless of the function executed on them. The only allowed leakage are the outputs, but even these are limited to the querying service that the owner(s) approved. We therefore assume that services will not submit malicious code. Handling these are beyond the scope of this work (and more generally – the topic of secure computation), but a

[1]It is also a bigger deterrent, as parties have real money at stake.

trivial improvement is to allow owners to specify more complex permissions, such as including hashes of approved functions, or to examine the amount of leakage using a privacy budget and if needed, add noise to the output using differential privacy [51]. Similarly, it is assumed that owners store valid data. Note that this does not mean we expect the owner to follow the sharing protocol, as this is easily discernible from the commitment scheme.

All other assumptions about the adversary stem from using the blockchain for consensus and broadcast. We assume a full-information model with a static (for short periods), rushing adversary. Namely, the adversary can see all messages and decide on its message in a round *after* seeing all the messages sent by the good players in that round. The adversary can also permute the ordering of the messages in each round. Parties interact using the blockchain's broadcast channel, and rely on it for consensus. Their communication normally occurs asynchronously in order to speed-up execution, but the upper bound is defined synchronously by the block time. To simulate private channels, we use PKI and assume the public encryption keys for each party are stored on-chain. Given the rationality assumptions of Section 6, we also assume that all messages are transmitted instantly through the off-chain broadcast (except for some negligible network propagation time), but are delayed by one round before they appear on the blockchain.

Guarantees. Given these assumptions, we can state the security and privacy properties the system provides, and under what conditions:

1. **Privacy and Correctness.** Owners are guaranteed w.h.p the long-term privacy of their data (see Section 12 for an analysis of data leakage over time). This is due to the security of secret sharing (with MPSS), MPC protocols and the assumption that on-chain contracts, such as randomly selecting quorums, are executed correctly. Similarly, services are guaranteed to obtain the correct outputs.

2. **Publicly Verifiable and Identifiable.** Since all commitments to inputs and transcripts of computations are public on the blockchain, any external party

can verify a computation and identify *all* cheaters without learning anything else.

3. **Financial Fairness[2].** (1) Services are guaranteed (correct) output, or their payment is returned. (2) Every honest party is rewarded[3], and each malicious party is penalized.

4. **Guaranteed Output.** If sufficient rational parties exist to complete the protocol, then output is guaranteed [4]. This was proven in Theorem 1b.

5. **(Nearly) Asynchronous MPC.** Rationality assumption of IC-MPC (see Section 6) also ensures that the execution is nearly asynchronous, since rational parties will not delay the protocol. The only delay that is cost-less for the adversary is to wait until just before the end of the current round before sending the output. The reason is that a party cannot participate in more than one computation per block, so it gains nothing by sending the output early in a round. So the delay is upper-bounded by a single round (compared to an $O(d)$ delay in fully synchronous MPC, where d is the depth of the circuit). However, since in practice the block-time is non-deterministic, even this delay becomes unlikely. Also note that by design there is no input-deprivation as data are shared in the pre-processing phase by owners.

Other security considerations and traces. Currently, the following traces of meta data and system-usage leak: the total amount of secrets stored; which owner (pseudonym) owns what data, though a different pseudonym could be used for each secret; what permissions an owner sets; pseudonyms of the computing parties; the functions to be evaluated; each computation request sent to the system; and which quorums store the secret (pseudonyms only). We note that our system could use its

[2]This is in contrast to previous work (e.g., [39]), where only traditional MPC protocols revealing the result to all parties were considered.

[3]Either by the service or by a malicious party. Either way, honest work is rewarded by the same amount.

[4]Contrast this with the standard infeasibility results of a dishonest majority. In this case, being malicious is not free, so even rational malicious parties would follow the protocol

own MPC capabilities to hide many of these traces, as well as other means such as [52] [7].

5 Online MPSS

Earlier, we intentionally omitted the implementation details of the re-sharing protocols. As mentioned before, for both security and integrity of the data stored in the system, secrets should not live forever in a single quorum, as an adversary could slowly corrupt these parties. The same logic applies to selecting the computation quorums, where a new quorum is selected for each computation. In this cases, we need a mechanism to re-share a secret securely from one quorum to the next. The basis for this re-sharing protocol is found in MPSS [22], and the presented protocol improves upon it. Intuitively, MPSS takes a secret x that lives in shares of a polynomial U in one quorum, and transfers it to another quorum, while atomically updating the polynomial to U'. The end result is that the original quorum Q only knows U and the new quorum Q' only learns U'. The old quorum then securely erases U, which succeeds assuming there are sufficiently many honest parties at the time of re-sharing.

For a full discussion of this protocol, we refer the reader to the cited paper, as only the necessary details to explain our improvements are provided. To create U' from U, parties in the old quorum collaborate to generate a distinct polynomial $U + V + W_k$ for each $p_k \in Q'$. Other than being random, V has the property that $V(0) = 0$ and for each W_k we have $W_k(k) = 0$. This provides the following relation:

$$\forall k \in Q' : U(k) + V(k) + W_k(k) = U(k) + V(k) = U'(k), \tag{7}$$

meaning that each party in the new quorum has a valid share of the secret, and that this share has been refreshed (due to V), invalidating the shares of the parties of the old quorum. The underlying protocol is quite expensive for real-time computations (recall that we do a re-sharing to a new computation quorum before each request), as it requires both a BA execution and several rounds to complete. We improve both of these significantly, by splitting the procedure into an online and an offline phase.

Without loss of generality, we show the process of a single re-sharing from Q to Q'.

Offline phase.

1. Q collectively generates (in batches) sets of $\{V + W_k\}_{k \in Q'}$.
    - Note that Q' is not known at this point and this is just a placeholder. We address it below.
2. The parties commit on-chain to these polynomials, where it is publicly verifiable that $V(0) = 0$ and $\forall k\ W_k(k) = 0$ hold.
3. Each $p_i \in Q$ locally stores its shares – $\{V(i) + W_k(i)\}_{k \in Q'}$ and commits to them on-chain.

Online phase. Starts with the blockchain requiring Q to re-share some U with Q'.

1. Each $p_i \in Q$ loads a pre-processed set and sets: $\forall k\ U'_k(i) = U(i) + V(i) + W_k(i)$.
2. $\forall p_i \in Q, p_k \in Q'$, p_i broadcasts $ENC(U'_k(i), p_k.epk)$.
3. Each $p_k \in Q'$ collects $m+1$ shares and reconstructs the polynomial to get $U'(k)$.
4. It then verifies its share using the on-chain commitments and disputes if something is wrong. A contract then checks this claim and penalizes the corrupted parties in Q (unless p_k lied, in which case it is penalized).

Notice that the online phase takes a single round. If enough parties are honest, there will be no reason to dispute. If they are not, they are discovered and penalized. An important thing to notice is that MPSS is not secure if parties in Q and Q' share the same identifiers. The authors in [22] used the public key of each party as their unique id. However, notice that this completely undermines our decoupled protocol, since in the offline phase, Q' is not known – i.e., the parties k's identifying the parties are not known. We address it by reserving the first $1, ..., |Q'|$ indices for the computation quorum.

6 Incentive-compatible MPC

The general-purpose MPC protocol defined earlier differs from previous work by adding a blockchain to the mix. The blockchain off-loads expensive consensus related parts of the protocol, publicly identifies corrupted parties, and handles incentives accordingly. So far, we have only explained how the protocol works, while providing some intuition into why it performs better than MPC in the standard cryptographic model, deferring dealing with rational agents to this section. Now, these assumptions and related results are formalized into the *incentive compatible multi-party computation* (or *IC-MPC*) model. We provide some initial results that are surprisingly strong (i.e., following the MPC protocol *honestly* is a strictly dominant strategy), showing that this model could extend beyond MPC to other variants of distributed and verifiable computing. We leave exploring this further for future work.

In a nutshell, IC-MPC extends MPC to the case of rational parties and incentives. It is inspired by how Nakomoto consensus works in Bitcoin [1] and is able to obtain a form of incentive-compatible BA (IC-BA) by incentivizing the miners to only agree on valid transactions. Note that IC-BA has never been formalized, but given the sheer amount of research around blockchain technology, there is an assumption that it works reasonably well. The IC-MPC model is in fact a generalization of this scheme. Currently, only cryptographic frameworks [67] [43] have been developed around blockchain and cryptocurrencies, so this work is the first to formalize an economic model as it relates to the security of a distributed system.

The main importance of IC-MPC is that it shows how rationality leads to better performance than MPC in the standard cryptographic model, if incentives exist. This is in contrast to previous work on rational MPC (e.g., [20], [21]), which yielded negative results on efficiency, namely – their protocols with rational parties were more restrictive and less efficient compared to the standard classification of parties as purely honest or malicious.

6.1 Definitions

Informally, IC-MPC requires three properties to work: (1) public identifiability of (actively) corrupt parties. (2) incentives distributed according to behavior. (3) Enough parties care about their rewards more than breaking the protocol. The first two properties are handled by the blockchain in practice, which we define formally below as an *observer*. The third property is an assumption on the utilities and that parties are rational. Notice that the definitions are general and not limited to MPC based on LSSS or implementing an observer using a blockchain (although we do assume a threshold adversary structure and leave general structures for future work). Formally, we define IC-MPC (with an observer) and RAND-IC-MPC (with a p-observer), a weaker notion that holds only on average, as follows:

Definition 1 (IC-MPC)**.** *A multi-party computation protocol with n parties, r of which are rational, is incentive compatible (or IC-MPC), if the dominant strategy for all rational players is to follow the protocol.*

Definition 2 (RAND-IC-MPC)**.** *If the dominant strategy for rational players is to follow the protocol in the expected case only, then we say the protocol is RAND-IC-MPC.*

Definition 3 (observer)**.** *An observer is a special mediator, that in each round (except the last) receives an input I_i^t, computes in $O(1)$ time $O_i^t := I_i^t$ if a party never deviated or $O_i^t := 0$ otherwise. In the last round T, $O_i^T = 1$ for honest parties and 0 otherwise.*

Definition 4 (p-observer)**.** *A p-observer is an observer who does not perfectly detect corruptions. Specifically, it correctly identifies honesty with probability $Pr[O_i^T = 1|i\ follows\ protocol] = p_1$ and similarly corruptions with probability $Pr[O_i^T = 0|i\ deviates] = p_2$*

The assumption on an observer running in $O(1)$ is used to model the fact that any meaningful cheater-detection is done in the post-processing phase, independent of the running time of the actual (online) computation.

6.2 Mechanism and Utilities

Equipped with these definitions, we ought to show that IC-MPC is a sensible concept. In other words, we need to construct a mechanism $(\Gamma, \vec{\sigma})$ for IC-MPC that satisfies the strong definition of having a $\sigma_i :=$ *follow the protocol* as a strictly dominant strategy (for all i). A mechanism boils down to developing a recommended protocol $(\vec{\sigma})$ that achieves some *good* outcome. The game Γ is defined by all possible ways parties execute the protocol – they can either follow it or arbitrarily deviate. Note that we have already defined the protocol in Section 4.5. Protocol 3 is the recommended σ_i for each party, whereas Contract 2 defines how the game is mediated by the observer.

For completeness, we re-state the essence of these protocols as an extensive-form game. In this game, every party i sends some input to the observer in every round t and the observer checks which players cheated. Except in the last round, the observer shares with all players the inputs of the parties who did not cheat, as they need it to continue the computation. The honest parties use this information to generate the inputs for the next round. In the final round $(t = T)$, the observer runs a final verification, and normalizes all outputs to be either 0 (malicious) or 1 (honest). The utilities of the players are defined solely by this boolean output vector, which marks whether a player should be rewarded or not. We assume that (rational) players prefer to be rewarded. Formally, these assumptions about the utilities are defined as follows:

- U1. If $O^T(r) = O^T(r')$ then $u_i(r) = u_i(r')$.

- U2. If $O_i^T(r) = 1$ and $O_i^T(r') = 0$ then $u_i(r) > u_i(r')$.

Note that these utilities are equivalent to [20] [21], but both our model and results differ greatly as mentioned earlier in this section. We can now prove two helpful lemmas.

Lemma 1. *Given a single game run r mediated by an observer and assuming U1, U2, the default strategy profile $\vec{\sigma}$ is the dominant strategy.*

Proof. Assume that this is not the case, i.e., wlog for some party i there are two game runs r, r', where i used the default strategy in r, but deviated and used a different strategy σ_i' in r', and this deviation was at least as profitable, namely – $u_i(r') > u_i(r)$. Since an observer perfectly identifies when a party deviates, we have that $O_i^T(r) = 1$ and $O_i^T(r') = 0$. This contradicts U2, and since U1 states that there are no other influences on i's utility, this is a contradiction. □

For the non-deterministic case, we have a slightly different result:

Lemma 2. *Given a game run r mediated by a p-observer and assuming U1, U2, the default strategy profile $\vec{\sigma}$ is the dominant strategy in expectation, if in addition $p_1 > p_2$ holds, where p_1 and p_2 are the probabilities of successfully classifying an honest or malicious party respectively.*

Proof. Obviously, the statement in the deterministic case does not hold, since a p-observer can mis-classify. However, here it is sufficient to show that U2 holds in expectation ($Eu_i(r) > Eu_i(r')$).

To show this, first observe that $p_1 = Pr[O_i^T(r) = 1|\ i\ follows\ protocol]$ and $p_2 = Pr[O_i^T(r) = 0|\ i\ deviates]$ are two random variables sampled from a Bernoulli distribution. Now assume as before that r, r' are games played using the default and a non-default strategies respectively. So the expected utility in r is $Eu_i(r) = p_1$, and similarly, the expected utility in r' is $Eu_i(r') = p_2$. Since we assumed $p_1 > p_2$, we have $Eu_i(r) > Eu_i(r')$ as required. □

6.3 Results

We are finally ready to prove the main results behind IC-MPC. These are simply the beneficial implications of having rational parties follow the default (honest) protocol.

Theorem 1. *Assume IC-MPC based on (n, t) secret-sharing, with n total parties, up to t of which are (actively) corrupted, r rational, and h are both honest and rational. Then,*

1. ***Efficiency.*** *Assuming an observer for validation and incentivization, if the underlying MPC protocol terminates successfully, it will run (nearly) as efficiently as a semi-honest execution.*

2. ***Correctness & Guaranteed output.*** *If $r > t$, the protocol will always terminate successfully.*

3. ***Privacy.*** *If $h \geq n - t$, privacy is guaranteed.*

Before we provide a sketch of a proof, we emphasize how these results improve upon MPC in the standard cryptographic model. Theorems (1a) and (1b) essentially mean that if enough parties are rational, namely – they care about earning (or at least not losing) money, all protocols will be optimally efficient with guaranteed *correct* output. So our model is able to capture the intuition of why cryptocurrencies are successful in achieving consensus at scale and replicate that in secure computation. In contrast, notice that theorem (1c) is not an improvement over the standard model. Since passive corruptions by definition are undetectable, we cannot penalize them. However, since we can assume overall less active corruptions (these are penalized), we can increase the secret sharing threshold to obtain overall better privacy guarantees. Theoretically, if *all* parties are rational, we could use $(n, n-1)$ secret sharing as no party will attempt to break protocol. In this case, it will be sufficient for a single party to be completely honest in order to preserve privacy. In practice, we still want to account for parties with unexpected utilities, as well as random byzantine faults, so we will use a slightly lower threshold.

Proof (Sketch). Theorem (1c) simply re-iterates the already known results, which our model cannot improve. The remaining results focus on active faults in the presence of rational parties. Theorem (1b) is trivially true from the assumptions on IC-MPC and the constraint on r. Since IC-MPC requires that following the protocol is the dominant strategy, which lemmas 1 and 2 satisfy, then at least r of the parties will execute the protocol correctly. Because $r > t$, there are sufficient parties to complete the protocol, regardless of what the other parties do. Theorem (1a) is also easy to

prove with the additional assumption of an observer. Recall that an observer mediates the protocol in $O(1)$ time per-round and verifies the execution. In other words, it does not affect the running time of the protocol, while still taking on the responsibility of a (potentially) expensive validation. The rational parties are therefore only required to execute an efficient semi-honest protocol (potentially with minor error-corrections and checks).

□

6.4 Asynchronous Communication and Repeated Games

As in cryptography, modeling an asynchronous game is not trivial, but as it turns out, the model of asynchronous secure computation [64] can be adapted to our model. This (cryptographic) model assumes up to t faults, and together with an assumption of eventual delivery and input deprivation, i.e., parties only wait for the first $n - t$ messages – whether they come from honest or malicious parties, the asynchronous computation can proceed in rounds similarly to the synchronous version.

In purely cryptographic models, active corruptions generally limit the threshold t more than passive corruptions. In other words, it is easier to protect against a passive adversary. Since the results of Theorem 1 intuitively specify that passive corruptions are harder in this model (after the correction below), it makes more sense to look separately at a rational adversary and an irrational one. For both adversaries, it is our goal to keep t, the secret-sharing threshold (or, *privacy threshold*), high as stated above (denote this as t_p). Only for the irrational adversary, should we protect against active corruptions (this threshold is marked t_r). To work in an asynchronous communication model, the protocol instructs honest nodes to wait in each round for $n - t_r$ messages. Note that we expect $t_r < t_p$ (less active corruptions), and this gap should increase as the number of rational parties the adversary controls grows. If all (corrupted) parties are rational then $t_r = 0$ and IC-MPC converges to the semi-honest model.

Theorem 1 is proven in a synchronous game-theoretical framework. Without modification, this implies that there is no penalty for the adversary if it waits to send the

corrupted parties messages until the very last minute (of the current round). In the context of the blockchain, the adversary can delay the evaluation of each layer of the circuit for minutes at a time. To overcome this, we can specify that the observer can get inputs in batches. Namely, an observer can get in each round (in the game) an input corresponding to several (MPC) rounds.

However, this does not incentivize adversarial parties to voluntarily process MPC rounds faster than the global clock of the game (and the observer). To enforce this, we need to look at the IC-MPC game as being played repeatedly. If we add an additional constraint that each party cannot play the game again before completing a previous execution, then rational parties (including corrupted ones) would attempt to complete the game run in as few rounds as possible, by sending the largest possible batch in each round. This constraint is easily enforced by the blockchain (in practice), or the observer, who will only select inactive parties for the quorum executing each instance of the game. This ensures that all rational parties attempt to complete the secure function evaluation in less than one round, which is also the maximum delay any rational (corrupted) party would impose. This leads to the following corollary.

Corollary 1. *In asynchronous IC-MPC, given that $n-t_r > t_p$ the maximum delay the adversary can impose is one round compared to a semi-honest asynchronous execution, while a synchronous MPC execution (even for the semi-honest case) has an $\Omega(d)$ lower-bound, where d is the depth of the circuit.*

7 Background: MPC over the Integers and Reals

This section marks the beginning of the second part of this thesis. We move away from the generic framework proposed earlier and consider the more interesting (but specific) case of secure computation over the integers and reals (in fixed-point representation). As will be discussed later, these are currently the main bottlenecks when trying to construct efficient secure protocols, for which support of integers/reals is required.

This section focuses on introducing notation and previously known results. In addition, the data representation of secret-shared values that encompass integers and

fixed-point numbers is explained in detail. These will be used as the basis for the optimized protocols that follow. The next two sections (Section 8 and 9) present improved solutions for fundamental protocols over the integers, while Section 10.4 focuses on improving performance over fixed-point numbers, through the example of optimizing division and normalization. Note that secure comparison is described in its own section, as it is arguably the most important building block after multiplication. Also, the ideas presented in that chapter are used later on as well.

7.1 Data Representation

As all secure protocols operate on elements of a finite field, it is required to find a suitable mapping of field elements to integers and reals.

Integers

Representing integers can be done directly. For a finite field of characteristic p, we represent $1, ..., \frac{p-1}{2}$ directly as their corresponding field elements. For the negative integers, we represent $-1, -2, ...$ as the sequence $p-1, p-2, ..., \frac{p+1}{2}$. The zero element is the same in the integers and over the field. Addition, subtraction and multiplication occurs over the integers in the same manner as it occurs over the field elements.

For other protocols such as comparison, we need to have a bit-wise representation of an integer. For a k-bit integer, this is done by keeping k shares – one for each bit. Each bit sharing holds an underlying secret in $\{0, 1\}$. Technically, any field larger than the number of parties is sufficient and a common choice is to use the extended field $\mathbb{F}_{2^8}$, as XOR can be done locally. For our applications we prefer representing the bits in the same field as the integers, as it requires less rounds of communication for converting back and forth between fields.

Fixed-point representation

Approximating real numbers is commonly achieved in one of two ways – floating point representation and fixed point. While the latter is less common in modern computing

architectures, after evaluating both options in the secure computation context [28], [33], it was clear that the better performing option is a fixed point representation. In Section 10.5, after presenting optimized building blocks, we revisit the notion of converting between representations to achieve the best of both.

Secure computation over fixed point numbers was first described in [28]. The data representation follows the idea that a fixed point number is an integer in disguise. It is simply an integer $\hat{x}$ accompanied by a scaling factor f, such that $x = \hat{x}2^{-f}$. Therefore, to secret share a real number x, we first set $fld(x, f) = trunc(x2^f)$ and share the result, which is an integer represented as a single field element. The inverse occurs when reconstructing, in order to obtain the result. With this defined, we can operate on real numbers by operating over field elements directly. The only remaining difference is that we need to ensure that the resolution f (which is public) is an invariant that is kept at the end of each secure protocol. We can use TruncPR (see below) to achieve that. We will use $k = e + f$ to describe the overall bit-length, where e is the range and f is the resolution. This representation allows us to represent numbers in the range $[-2^{e-1} + 2^{-f}, 2^{e-1} - 2^{-f}]$, in 2^{-f} intervals.

Notation

For clarity of the secure protocols presented in this and the next sections, we use the following notation:

- x represents a public integer/fixed-point number.
- $[x]$ or $[x]_p$ represents the set of shares collectively held by the parties. An operation on $[x]$ implies that each party performs that protocol on its own share. For example, $[c] \leftarrow [a][b]$ implies that all parties run the multiplication protocol to obtain shares of the result c. In general, the field $\mathbb{Z}_p$ will remain implicit, unless the protocol requires working in multiple fields, in which case the explicit notation $[x]_p$ is used.
- $[x]_B := ([x_{k-1}], ..., [x_0])$ represents a bit-wise sharing. The short-hand notation $[x]_B$ will be frequently used to describe the entire set, while $[x_i]$ refers to a single

bit in the set.

- $[x]_i := ([x_{m-1}], ..., [x_0])$ represents a single block of bits. In some cases, a k-bit integer is split to several m-bit blocks. The outer subscript marks the block index, while the inner subscript marks the bit index within that block. For example, the j-th bit in the i-th block is written as $[x_j]_i$ (or x_{ji} if x is public).
- $[x^{(i)}]$ (or $x^{(i)}$ if public) – a superscript is used to distinguish different variables that have some shared contextual meaning. For example, if we need to generate several random values, then a superscript will be used to index them.

7.2 Existing and Elementary Building Blocks

Beyond addition, multiplication and open (or reveal), there are several well known elementary operations that we use. Below is a list of protocols that we use internally. For completeness, we also include protocols which are already very efficient and therefore were not our focus.

Simple Functions

The following are a list of elementary MPC protocols and a list of utility functions.

- **Addition.** $[c] \leftarrow [a] + [b]$.
- **Multiplication.** $[c] \leftarrow [a][b]$ (1 rnd, 1 inv).
- **Open.** $a \leftarrow Open([a])$ (1 rnd, 1 inv).
- **Multiply and Open.** $[c] \leftarrow MulPub([a], [b])$ (1 rnd, 1 inv).
- **Bits.** $a_B \leftarrow Bits(a, k)$. Returns the first k-bits of a public value a.
- **Blocks.** $(a_{\lceil \frac{k}{m} \rceil - 1}, ..., a_0) \leftarrow Blocks(a, m)$. Returns $\lceil \frac{k}{m} \rceil$ blocks of m bits each of the decomposed public a.
- **Next Prime.** Getting the next prime after an integer n is done via $p \leftarrow NextPrime(n)$.

- **Field Mapping.** As shown above, $\hat{a} \leftarrow fld(a, f)$ maps an integer/real to a field element with resolution f.

Pre-processed Randomness Protocols

Generating random field elements and integers can be achieved in multiple ways. For this work we make use of the protocols in [25], in addition to pseudo-random secret sharing (PRSS) and pseudo-random integer sharing (PRIS) [38] [58].

- **RandF().** Returns a random field element.
- **RandB().** Returns a random bit using the protocol from [25]
- **RandN(k).** Returns a random integer in the range $[0, 2^k)$.
- **RandInv().** Based on [73], we can generate a random invertible element and its inverse. This is later used to create RandExpInv, which is used internally in the pre-processing of Protocol 30.

Composing these protocols, we obtain other helpful functionalities –

- **RandNB(k, m).** Returns a random integer in the range $[0, 2^k)$ and a sharing of the first m bits. This is trivially achieved by generating k random bits and setting $[r] \leftarrow \sum_{i=0}^{k-1} 2^i[r_i]$.
- **RandDF(p, q).** This protocol returns the same random field element in two different fields. It works by creating a random field element in $\mathbb{Z}_p$, then invoking the share conversion protocol below to create a sharing in $\mathbb{Z}_q$.

Unbounded Fan-in Multiplication and Prefix Multiplication

Using the method of Bar-Ilan and Beaver [73] with the improvement by [30], one can compute in one round and linear communication the multiplication of k shared non-zero secrets. Moreover, we can compute the prefix product of these k inputs. Formally, given $[a1], ..., [ak] \in \mathbb{Z}_p$, the prefix product is defined as:

$$[p_j] = \prod_{i=1}^{j} [a_i], \forall j \in [1..k] \tag{8}$$

The main idea behind this protocol is to generate random, invertible shared field elements and their inverses $[r_i], [r_i^{-1}]$. Then, set $[s_i] = [r_{i-1}^{-1}][r_i]$. Note that up to this point there are no dependencies on the data so these random elements can be efficiently pre-processed.

In the online phase, we multiply and reveal each $m_i := [s_i][a_i]$, taking 1 round and k invocations. To obtain the prefixes, we locally compute:

$$[p_j] = [r_j^{-1}](\prod_{i=1}^{j} m_i), \forall j \in [1..k], \tag{9}$$

which evaluates to the formula above.

Share conversion

Based on [37], Protocol 4 illustrates how to convert polynomial shares from one finite field to another. The protocol is statistically secure and takes a single round and a single invocation.

Algorithm 4: ConvertZp2Zq($[x]_p, q$)

$([r]_p, [r]_q) \leftarrow RandDF(p, q)$;

$c \leftarrow Open(2^k + [x]_p + [r]_p)$; `// 1 rnd, m inv`

$c' \leftarrow c - 2^k$;

$[x]_q = c' - [r]_q$;

return $([x]_q)$

Mod2

Mod2 (based on [29]) shown in Protocol 5 is used to extract the LSB of a secret shared integer. The protocol is statistically secure and costs 1 round and 1 invocation (a single opening). This protocol is optimally efficient, which is only possible since it is statistically private.

Algorithm 5: Mod2($[x], k$)

$([r], [r]_B) \leftarrow RandNB(k+1, k)$;

$c \leftarrow Open(2^k + [x] + [r])$;

$[x_0] \leftarrow c + [r_0] - 2c_0[r_0]$;

return $[x_0]$

TruncPR

TruncPR is a statistically accurate protocol for truncating a secret integer where the last truncated bit is probabilistically rounded up or down. This protocol, shown in [29], introduces a small potential error, while avoiding an expensive secure comparison. Our secure comparison protocol based on 8 presents similar online efficiency, but for applications that do not require a deterministic result, using this variant is still preferable.

Algorithm 6: TruncPR($[x], k, m$)

$([r], [r]_B) \leftarrow RandNB(k+1, m)$;

$c \leftarrow Open(2^k + [x] + [r])$;

$c \leftarrow c \ mod \ 2^m$;

$[r'] \leftarrow \sum_{i=0}^{m-1} 2^i [r_i]_B$;

$[x'] \leftarrow ([x] - c + [r'])(2^{-m} \ mod \ p)$;

return $[x']$

Note that this protocol truncates m bits like the accurate protocol, namely – returns $[x'] = \lfloor \frac{[x]}{2^m} \rfloor + u$ (u is the rounding bit). The rounding bit in this case is determined statistically due to the relation $Pr[u = 1] = Pr(r' + a' \geq 2^m)$.

FPMul

Fixed-point multiplication [28] is an extension for the regular integer multiplication protocol. It uses the fixed-point share representation and then truncates the added f bits to scale the result back to a fixed-point representation with f precision. The protocol uses TruncPR for efficiency, since the result is inherently an approximation.

Algorithm 7: FPmul($[a], [b], k, f$)

$[c] \leftarrow [a][b]$;

$[d] \leftarrow TruncPR([c], 2k, f)$;

return $[d]$

Sign Modules and CMP_k

Recently, Yu [66] presented the idea of *sign modules*. Intuitively, a sign module is defined over a finite field $\mathbb{Z}_p$ that maintains that for every $(x \ mod \ p) \in \{1, ..., k\}$ is a quadratic residue in the field, and equivalently every $(x \ mod \ p) \in \{-1, ..., -k\}$ is a quadratic non-residue. The authors present a randomized algorithm for finding a prime $p \sim 2^l$ with a $\Omega(k)$ sign module. With this idea of a sign module, the authors construct a protocol for comparison of small integers (up to $|k| > \log p$), which is then used to create an improved version of equality tests and elementary boolean gates and functions. Note that these results can be achieved with the same online complexity (and at times - better) using our approach of generating look-up tables, as shown in Section 8. The pre-computed tables have the benefit of not restricting the finite field, but require more pre-processing. They also have an additional benefit of being able to handle comparison of small (yet still larger) numbers. Most of our protocols make use of both of these in a way that leverages the best of both. However, if sign modules cannot be used (for example, if we are using a specific finite field or require a composite), then any call to CMP_k can be assumed to be implemented using look-up tables. Then, any protocol that is composed using CMP_k, i.e., BitsEQ, OR/AND, Prefix OR/ Prefix AND and bit-wise comparison, would have the same online complexity with slightly larger pre-processing.

The main result sign modules gives us is CMP_k shown in Protocol 8.

Algorithm 8: $CMP_k([x], [y])$

$[a] \leftarrow RandF()$; $[b] \leftarrow RandF()$;

$[c] \leftarrow [a][a]$;

$[d] \leftarrow [a][b]$;

$[e] \leftarrow [c][d]$;

$f \leftarrow Open([d][d])$; `// Abort if` $f = 0$

$[r] \leftarrow (\sqrt{f})^{-1}[e]$;

$[s] \leftarrow (\sqrt{f})^{-1}[d]$;

$[z] \leftarrow [x] - [y]$;

$z' \leftarrow Open((2[z] + 1)[r])$; `// 1 rnd, 1 inv`

$[u] \leftarrow LegendreSymbol(z')[s]$;

$[v] \leftarrow ([u] + 1)(2^{-1} \ mod \ p)$;

return $[v]$

This protocol pre-processes a random $[r]$ and its sign $[s]$, where the sign (in a sign module) is simply defined as the Legendre symbol. Using this, in the online phase a single value is revealed. This is secure since the inputs are masked by a multiplicative mask $[r]$, which does not leak any information as long as the input is non-zero. This is ensured by using $2[z] + 1$ instead of $[z]$ directly. The next step is to compute the sign of $[z]$ by multiplying the sign of z' (i.e., its Legendre symbol) by the pre-computed sign of r. The final step simply maps the result to $\{0, 1\}$. The cost is therefore a single round with a single invocation (one opening).

Equality Test

Note that CMP_k preserves the sign of at least all integers up to $|\log(p)|$. With this in mind, we can efficiently solve equality and some other symmetric boolean functions by counting the numbers of bits that are equal to one in a field element. By definition these are guaranteed not to exceed $\log(p)$.

For equality, this translates to Protocol 9 (for the bitwise case) and to Protocol 10 for the integer case. The latter is as efficient (online) as the protocol in [27], but more efficient offline as it requires constant work in that phase as well. However, it

adds a constraint on the value of p. In addition, the bitwise case is not easily reduced from [27], and proves to be more widely applicable in sub-protocols.

Algorithm 9: BitsEQ$(c, [r]_B, k)$

$[d] \leftarrow \sum_{i=0}^{k-1} c_i + [r_i] - 2c_i[r_i]$;

$[b] \leftarrow CMP_k(0, [d])$; `// 1 rnd, 1 inv`

return $[b]$

Algorithm 10: EQ$([x], [y], k)$

$([r], [r]_B) \leftarrow RandNB(k+1, k)$;

$c \leftarrow Open(2^k + [x] - [y] + [r])$; `// 1 rnd, 1 inv`

$[b] \leftarrow BitsEQ(c, [r]_B, k)$; `// 1 rnd, 1 inv`

return $[b]$

OR, AND

OR and AND gates are similar to bit-wise equality, but instead of counting the number of ones and comparing them to zero, we do a comparison against one (for OR) or k (for AND).

Algorithm 11: OR$([x]_B, k)$

$[b] \leftarrow CMP_k(\sum_{i=0}^{k-1}[x_i], 1)$; `// 1 rnd, 1 inv`

return $[b]$

Algorithm 12: AND$([x]_B, k)$

$[b] \leftarrow CMP_k(\sum_{i=0}^{k-1}[x_i], k)$; `// 1 rnd, 1 inv`

return $[b]$

Prefix OR, Prefix AND

To do prefix operations, we can invoke OR and AND in parallel for each bit. This improves upon the best known results to date used in [29], as it requires only a single round.

8 Efficient Secure Comparison

While addition and multiplication are sufficient for completeness theorems [6], many of the most basic protocols cannot be constructed simply by direct arithmetic of secret values in a finite field. These require working with a bit-representation of the values, which ultimately boils down to executing one or more secure comparison protocol of shared bits. Therefore, constructing a more efficient secure comparison protocol would have widespread implications for a large portion of other secure protocols that use it as a sub-protocol. Essentially all operations that work over the integers or reals, but cannot be likened to field operations, rely on comparisons.

Compared to previous research, the concern of this work is practical efficiency rather than asymptotic. In this section we present a series of efficient secure comparison protocols, each is specialized for a specific range of k (the bit-size of the inputs). Our main observation is that most applications of interest are focused on the 32-bit/64-bit case, and therefore we start by optimizing this scenario and obtain an optimally efficient protocol that only has 1 round and 1 multiplication (online) for bit-wise comparison. Using the ideas developed for this the case of 'small' (i.e., up to 64-bit) integers, we develop solutions for larger inputs that are up to thousands of bits. These solutions are bundled together in a hybrid bit-wise comparison protocol (Protocol 13), that selects the best approach depending on the size of k. Also, while it is not likely to see applications that require larger inputs than that (as the inputs are exponential in the bit-length), we also provide an asymptotically efficient solution with the same asymptotic complexity as the current state-of-the-art [27], but with improved constants.

For well over a decade, a significant body of work has been dedicated to improving secure comparison. Until recently, all secure comparison protocols required bit-decomposition or bit-wise protocols. The first such solution was proposed in [25] and later improved by [26], but were highly in-efficient in practice. In [26], the authors were able to reduce the comparison problem to that of bit-wise comparison of a public and secret value. This was later used to create more efficient solutions, and

most protocols today utilize this reduction (including ours). Therefore, our evaluation through-out this section focuses on the bit-wise comparison of a public and secret value.

Although these solutions required constant rounds in theory, in practice the actual constants were large and the communication (linear in k with high constants) was prohibitive. As a result, some alternatives (e.g., [54], [28]) considered log-depth circuits with less communication and rounds in practice. One of the most competitive (practical) solutions to date was developed by [30] and later improved in [29], while settling for statistical security. This solution requires only 2 rounds and $k + 1$ communication. While the constants are nearly optimal, the linear communication is still prohibitive, and a recent survey of these different techniques [55] has shown that in practice, the best protocol depends on the use-case. For the asymptotic case, the only constant-round solution with sub-linear communication complexity to date was presented by Toft. et al [27], but for practical applications it is likely less efficient.

As we show in this section, all of these results can be significantly improved. When considering actual values of k that are used in reality, we show that much better solutions exist that allow for both low round complexity and low communication, presuming some additional pre-processing is done and statistical security is sufficient. The results are summarized in Tables 1 and 1. Protocol 13 shows the hybrid protocol, which simply points to the appropriate underlying algorithm optimized for the given inputs. The parameter m will be explained below, but in practice is globally set to $m = 8$, which would call BitGTS if $k \leq 64$, BitGTM for $k \leq 512$ and BitGTL otherwise.

Source	Rounds	Communication	Security
[25], [26]	6	9k	Perfect
[29]	2	k+1	Statistical
[27]	14	$20\sqrt{k}$	Statistical
This work	1	1	Statistical

Table 1: Online complexity of secure bitwise comparison protocols (up to 64-bit integers and for $m = 8$)

Source	$k \leq 512$		$k \leq 4096$		Asymptotic		Security
	Rounds	Comm	Rounds	Comm	Rounds	Comm	
[25], [26]	6 rounds, 9k invocations						Perfect
[29]	2 rounds, k+1 invocations						Statistical
[27]	14 rounds, $20\sqrt{k}$ invocations						Statistical
This work	2	9	3	73	4 to 8	$O(1)$ to $2\sqrt{k}+67$	Statistical
			5	$\sqrt{k}+18$			

Table 2: Online complexity of secure bitwise comparison protocols (larger values and for $m = 8$)

Algorithm 13: HybridBitGT$(c, [r]_B, k, m)$

if $k \leq m^2$ **then**

  $\{\hat{r}_i\}_{i=1}^{m} \leftarrow$ load look-up tables for $[r]_B$;

  $[b] \leftarrow BitGTS(c, \{\hat{r}_i\}_{i=1}^{m}, m)$; `// See Protocol 16`

if $m^2 < k \leq m^3$ **then**

  $\{\hat{r}_{ij}\}_{ij=(1,1)}^{\lceil \frac{k}{m^2} \rceil, m} \leftarrow$ load look-up tables for $[r]_B$;

  $[b] \leftarrow BitGTM(c, \{\hat{r}_{ij}\}_{ij=(1,1)}^{\lceil \frac{k}{m^2} \rceil, m}, m, \lceil \frac{k}{m^2} \rceil)$; `// See Protocol 17`

if $k > m^3$ **then**

  $[b] \leftarrow BitGTL(c, [r]_B, k, m)$; `// See Protocol 18`

return $[b]$

In this section, as well as the next two sections that deal with specific secure protocols over the integers/reals, each protocol will include a correctness, security and complexity analysis. All protocols exhibit either perfect or statistical security. We will focus on the semi-honest case for clarity and ease of comparison to previous protocols. Adjusting for the active (more accurately – rational) case can be done using the general framework presented earlier in this thesis, where owners first commit to their inputs on the blockchain and parties provide their computation transcript as proof of correctness, and are penalized or paid based on their behavior.

8.1 Comparison of Small Inputs ($k \leq 64$)

The most common case in modern computing systems is the use of 32/64-bit representations. With the exception of cryptographic and a small number of arbitrary precision applications, both hardware and software have been optimized to utilize exactly these cases. Therefore, a good first step is to construct an optimized protocol for this case specifically, where the bit-length $k \leq 64$. Note that this protocol does not have to be asymptotically optimal, and in fact – it would have a $d = O(log_m k)$ round complexity in some parameter m described below if it is applied recursively. However, as it will not be used directly for larger input sizes, we will only consider the case where $d = 1$ in this subsection, or $d = 2$ in the next. Our requirement is therefore that this protocol is optimal for integers with bounded bit-length that in practice cover most applications of interest.

Recall that computing a comparison of two shared integers could be reduced to a bit-wise comparison of a public value (c) and a pre-computed shared $[r]_B$. With this in mind, it is possible to pre-process a look-up table mapping from each c directly to the secret solution $[c < r]_B$. While secret indexing is generally expensive, public indexing and the pre-processing of $[c < r]_B$ is free in the online phase. For the offline phase, we pre-process the lookup tables based on Protocol 14. This naturally leads to an unfavorable asymptotic complexity of $O(2^k)$, so instead we pre-compute small look-up tables by constraining the exponent m to satisfy – $2^m \sim k$. This relation should hold except for some small constant factor.

In practice, for $k \leq 64$ we have that $m = 8$ satisfies this constraint, as we obtain $2^8 = 4 \cdot 64$. This is also the parameter used to achieve optimal results of 1 round and 1 multiplication, but the protocols presented are parametrized by m for generality.

Algorithm 14: GenTable($[r], [r_m], ..., [r_1], m$)

$\hat{r} \leftarrow EmptyDict()$;

for *c:=1,...,*2^m **do**

  $[b'_c] \leftarrow BitGT'(c, ([r_m], ..., [r_1]))$; `// 1 rnd, m+1 inv`

  $[d_c] \leftarrow BitsEQ(c, ([r_m], ..., [r_1]))$; `// 1 rnd, 1 inv`

  $[b_c] \leftarrow [b'_c] - (1 - [b'_c])(1 - [d_c])$; `// 1 rnd, 1 inv`

  $\hat{r}.set(c, [b_c])$;

return $\hat{r}$

Algorithm 15: GenAllTables(k, m)

$([r], [r_k], ..., [r_1]) \leftarrow RandNB(k, k)$;

for *i:=1,...,*$\lceil \frac{k}{m} \rceil$ **do**

  $\hat{r}_i \leftarrow GenTable(\sum_{j=1+(i-1)m}^{m+(i-1)m} [r_j], [r_{m+(i-1)m}], ..., [r_{1+(i-1)m}], m)$;

return $\hat{r}_{\lceil \frac{k}{m} \rceil}, ..., \hat{r}_1$

To complete the offline phase, Protocol 15 accepts the bit-lengths k, m as an input and generates the shared randomness and the derived look-up tables for each $\frac{k}{m}$ block. Generating the tables amounts to splitting a given $[r]_B$ into m-bit blocks, and then enumerating for each block all 2^m possibilities for c, and using CMP_k or a similar bit-wise comparison protocol (e.g., the one in [29]), with one difference. Instead of returning for each c shares of the bit $[c > r]$, the result is in $\{-1, 0, 1\}$, where 1 is set when $c > r$, 0 denotes equality and -1 marks that $c < r$. Furthermore, note that the original $[r]_B$ can be discarded and only a local pointer $\hat{r}_i$'s (one for each block) needs to be retained in its place. There are additional optimizations to this naive brute-force approach that would reduce the amount of pre-computation and storage. Most notably, instead of enumerating every single possibility for c, we could sample each $l - th$ value. For the case of $m = 8$, we could set $l \in \{4, 8\}$, which would reduce the amount of pre-processing (and storage) required for each table from 256 values to (64, 32) respectively. For reasons that will soon become clear, the resulting secret entry in the table should be multiplied by the sampling rate l. Namely, each entry is in $\{-l, 0, l\}$ (for simplicity we would maintain that $l = 1$ from here-on). Other optimizations such as generating the look-up values in an order that minimizes

repetitions could further boost performance.

Algorithm 16: BitGTS$(c, \{\hat{r}_i\}_{i=1}^{m}, m)$

$(c_m, ..., c_1) \leftarrow Blocks(c, m)$;

for *i:=1,...,m* **do**

  $[d_i] \leftarrow \hat{r}_i.get(c_i)$;

$[d] \leftarrow \sum_{i=1}^{m} 2^{i-1}[d_i]$;

$[b] \leftarrow 1 - CMP_k([d])$; `// 1 rnd, 1 inv`

return $[b]$

We can now turn our attention to the online BitGTS protocol. Shown in Protocol 16, the algorithm accepts as an input a public c and the set of look-up tables $\{\hat{r}_i\}_{i=1}^{\lceil \frac{k}{m} \rceil}$. An additional constraint introduced for this protocol is that $k \leq m^2$. Taking the worst-case into consideration, we simplify the expression $\lceil \frac{k}{m} \rceil = m$. An analysis of the protocol is brought below.

Correctness. The protocol begins with each party locally splitting c into (at most) m blocks of size m-bits (due to the constraint above). Then, using each c_i as an index to the associated $\hat{r}_i$, we obtain $[d_i] \in \{-1, 0, 1\}$ as described above. In the next step, all of the $[d_i]$'s are locally added together to construct an m-bit shared integer, between $-2^{m-1} \leq d \leq 2^{m-1}$. For the more general case of $l > 1$, the result is an $(m + \log_2 l)$-bit integer instead.

Note that if d is negative, then $c < r$ and vice versa. Also, $d = 0$ when $c = r$. Thus, the problem is reduced to checking the sign (i.e., comparison to zero) of a small number. Since we constrained $k \sim 2^m$, This could be done in a single round and in one multiplication using CMP_k. Another alternative is to use an additional pre-processed look-up table, this time with entries in $\{0, 1\}$, to do the comparison. However, as this would also require to reveal one value (which has the same cost), as well additional pre-processing, calling CMP_k is preferred. Only in situations where the sampling rate l is chosen to be large enough, we would favor selecting a smaller prime for the field and use an additional and slightly larger look-up table instead.

Security. Security follows from the fact that the protocol includes only local operations over secret shares and public values, which by definition does not reveal

anything new, and well-defined sub protocols that are secure. The pre-processing step is also composed by calls to secure sub-protocols and generates results that are oblivious to the actual data which is not yet known in this stage.

Complexity. For the case of $k \leq 64$ covered here, which covers most applications, the complexity of the protocol in the online phase is optimal – 1 round and 1 multiplication. This holds since all interactions have been completely eliminated using pre-processing and local operations in the online phase. The only interaction is required for computing a comparison of a small integer which can be done in a single round and one invocation of the multiplication protocol. For the general case where we compare two secret shared values in the field, $[x]_p, [y]_p$, an additional 1 round and 1 multiplication is required.

8.2 Comparison of Medium Inputs ($k \leq 512$)

For medium-sized integers, Protocol 17 is given. Note that it is functionally equivalent to the previous protocol, but instead of relying on look-up tables as its base-case protocol, it calls BitGTS[5]. In other words, BitGTM splits its inputs into blocks that BitGTS can handle and calls it recursively exactly once. Then, BitGTS splits its inputs to blocks the look-up tables can process and returns the result. Blocks are combined in the same way and tested efficiently using the small comparison protocol CMP_k.

Adding an additional constraint that $1 < \beta \leq m$, we obtain that this protocol is efficient for $m^2 < k \leq m^3$. The case of $k \leq m^2$ (i.e., $\beta \leq 1$) is already covered by BitGTS directly.

[5]We use a slightly different version here that returns a value in $\{-1, 0, 1\}$

Algorithm 17: BitGTM($c, \{\hat{r}_{ij}\}_{ij=(1,1)}^{\beta,m}, m, \beta$)

$(c_\beta, ..., c_1) \leftarrow Blocks(c, m^2)$;

for *i:=1,...,β (in parallel)* **do**

  $[d_i] \leftarrow BitGTS(c_i, \{\hat{r}_{ij}\}_{j=1}^{m})$; `// 1 rnd, β inv`

$[d] \leftarrow \sum_{i=1}^{\beta} 2^{i-1}[d_i]$;

$[b] \leftarrow 1 - CMP_k([d])$; `// 1 rnd, 1 inv`

return $[b]$

Correctness and security follow from the same arguments of BitGTS and the fact that this protocol calls BitGTS as a sub-protocol.

Complexity. The offline storage and communication complexity is proportional to β times that of BitGTS (same round complexity). Online complexity involves one round with β multiplications and an additional round with one multiplication for the last comparison (a total of 2 rounds, $\beta + 1$ multiplications). For our choice of $m = 8$, this implies 2 rounds and 9 multiplications for comparison of the maximum 512-bit integers.

8.3 Comparison of Large Inputs ($k \leq 4096$)

Note that BitGTS and BitGTM are instances of a $d = O(log_m k)$ rounds protocol with $d \in \{1, 2\}$. Both gain considerable efficiency boost as a result of the pre-processing of look-up tables and the idea that k cannot be too large in practice, since inputs are exponential in the bit-length. While we could use the same idea for larger k's, we note that the amount of pre-processed tables increases linearly in k asymptotically. To observe this, note that $\lceil \frac{k}{m} \rceil$ tables are needed and for $k >> m$, this implies $O(k)$ work.

Although this is reasonable asymptotically, for practical applications when $d > 2$, but when d is still small, the number of pre-computed tables becomes large enough that an alternative protocol should be considered – one that trades-off having an additional (constant) number of rounds but does not increase the pre-processing cost beyond that of a single BitGTM call. This variation (BitGTL) is shown in Protocol

18.

Assuming as before $m = 8$, then the question whether BitGTL (more rounds) is preferable to recursively calling the previous protocols (more pre-processing) seems to depend on other externalities for very large integers (e.g., $k = m^4 = 4096 \rightarrow d = 3$). For the framework in this thesis the choice was to transition to the more rounds solution when $k > m^3$.

Algorithm 18: BitGTL$(c, [r]_B, k, m)$

$(c_\beta, ..., c_1) \leftarrow Blocks(c, m^3)$; // $\beta = \lceil \frac{k}{m^3} \rceil$

$([r]_\beta, ..., [r]_1) \leftarrow Blocks([r]_B, m^3)$;

for *i:=1,...,β (in parallel)* **do**

  $[d_i] \leftarrow 1 - BitsEQ(\sum_{j=1}^{m^3} 2^{j-1} c_{ji}, \{[r_j]_i\}_{j=1}^{m^3})$; // 1 rnd, β inv

$[e]_B \leftarrow PrefixOR([d_\beta], ..., [d_1])$; // 1 rnd, β inv

for *i:=1,...,β − 1* **do**

  $[f_i] \leftarrow [e_i] - [e_{i+1}]$;

$[f_\beta] \leftarrow [e_\beta] - 1$;

$[\tilde{c}] \leftarrow \sum_{i=1}^{\beta} [f_i] c_i$;

$[\tilde{r}] \leftarrow \sum_{i=1}^{\beta} [f_i][r_i]$; // 1 rnd, β inv

$[b] \leftarrow GTM(\sum_{j=1}^{m^3} 2^{j-1}[\tilde{c}_j], \sum_{j=1}^{m^3} 2^{j-1}[\tilde{r}_j], m)$; // 3 rnd, $m^2 + 2$ inv

return $[b]$

Correctness. BitGTL takes a somewhat different approach from BitGTM/BitGTS. Both are designed to first reduce the problem of comparing k-bit integers to that of k'-bit ones, where $k' < k$. However, the process that leads to achieving that varies. In this protocol, the idea is to first split c and $[r]_B$ to $k' \leq m^3$-bit blocks. Then, unlike the previous protocols, which solve each block independently and then compose the solutions, we first find the most significant block where c and r differ (since that is where the comparison relation is determined). This is achieved by a series of equality tests of each block in parallel as suggested in [27]. The (negation) of each equality result is stored in $[d_i]$, $i \in \{1, ..., \beta\}$ where $\beta = \lceil \frac{k}{m^3} \rceil$. In other words, the first i, such that $[d_i] = 1$ points to the location of the (only) block we need to compare in practice.

The next step involves a Prefix OR operation (stored in $[e]_B$), then by locally

setting each $[f_i] \leftarrow [e_i] - [e_{i+1}]$, we obtain an all zeroes vector except for 1 in the correct block. Using this, we can obliviously select the correct block for c and $[r]_B$ by point-wise multiplying each block and summing the results up in $[\tilde{c}]$ and $[\tilde{r}]$. The result is that both of these would include m^3-bit numbers taken from the single (correct) block. The final step is therefore to call GTM once and obtain the result, where GTM is the non-bitwise version of BitGTM (implies an additional 1 round and 1 multiplication).

Security. All non-local operations are either multiplications (including one prefix multiplication) or calls to BitsEQ and GTM protocols which have been proven to be secure. Since the protocol does not publicly open any element, except through the usage of these secure sub-protocols, then no new information can be learned.

Complexity. Using $\beta = \lceil \frac{k}{m^3} \rceil$ and m as parameters, the runtime complexity is 5 rounds and $m^2+2\beta+2$, where β is a small constant for any practical application and m is small by definition. For the choice of $m = 8$ as before, we can observe that even for very large integers (e.g., $k = 4096$), BitGTL requires an order of $\sqrt{k}$ communication, but unlike the sub-linear protocol of [27], the constants are significantly lower.

For the asymptotic case, the communication complexity becomes linear in k, since $k >> m^3$, which is not an improvement over known results. However, it is unreasonable to assume that such applications for computing over encrypted data exist in practice, as for a choice of $m = 8$ an asymptotically better algorithm would only perform better on bit-lengths of at least 10^5.

8.4 Asymptotic Complexity

As mentioned, for virtually all applications and assuming we set $m = 8$ (or some similar value), we can solve the secure comparison problem significantly more efficiently than any existing asymptotically optimal solution, with one of the protocols – BitGTS, BitGTM or BitGTL. These protocols are efficient for bit-lengths of up to m^2, m^3 and $\sim m^4$. Given $m = 8$, BitGTL only becomes less efficient in a situation where $k >> 4096$, which is unlikely. Nevertheless, we illustrate for completeness how protocol BitGTL could be adjusted to support a constant round solution for

the asymptotic case with sub-linear communication complexity. While the changes are small we differentiate the two by describing the asymptotic protocol as BitGTA (Protocol 19). Note that the asymptotic protocol is not part of the hybrid protocol, as for practical applications BitGTL is better equipped to handle any sized integer. These adjustments make the protocol a high-level equivalent of the constant round comparison protocol of [27], so the two share the same asymptotic complexity, but with lower constants due to the efficient constructions presented earlier. We present a sketch of the required changes below.

Algorithm 19: BitGTA$(c, [r]_B, k, m)$

$(c_{\sqrt{k}}, ..., c_1) \leftarrow Blocks(c, \sqrt{k})$;

$([r]_{\sqrt{k}}, ..., [r]_1) \leftarrow Blocks([r]_B, \sqrt{k})$;

for *i:=1,...,*$\sqrt{k}$ *(in parallel)* **do**

  $[d_i] \leftarrow 1 - BitsEQ(\sum_{j=1}^{\sqrt{k}} 2^{j-1} c_{ji}, \{[r_j]_i\}_{j=1}^{\sqrt{k}})$; `// 1 rnd,` $\sqrt{k}$ `inv`

$[e]_B \leftarrow PrefixOR([d_{\sqrt{k}}], ..., [d_1])$; `// 1 rnd,` $\sqrt{k}$ `inv`

for *i:=1,...,*$\sqrt{k} - 1$ **do**

  $[f_i] \leftarrow [e_i] - [e_{i+1}]$;

$[f_{\sqrt{k}}] \leftarrow [e_{\sqrt{k}}] - 1$;

$[\tilde{c}] \leftarrow \sum_{i=1}^{\sqrt{k}} [f_i] c_i$;

$[\tilde{r}] \leftarrow \sum_{i=1}^{\sqrt{k}} [f_i][r_i]$; `// 1 rnd,` $\sqrt{k}$ `inv`

$[b] \leftarrow HybridGT(\sum_{j=1}^{\sqrt{k}} 2^{j-1}[\tilde{c}_j], \sum_{j=1}^{\sqrt{k}} 2^{j-1}[\tilde{r}_j], m)$; `// 2-6 rnd,` $O(1) - O(\sqrt{k})$ `inv`

return $[b]$

The main idea is to split c and $[r]_B$ into $\sqrt{k}$-bit blocks (see [27] for details) instead of m^3-bit. This ensures that the number of blocks is sub-linear, as they are dependent on k and not on some exponent of a constant parameter. This leads to reducing the problem from a comparison of k-bit integers to that of $\sqrt{k}$-bit ones, which we can solve with a single call to to the HybridGT protocol. Depending on how large $\sqrt{k}$ is, HybridGT solves the problem in a constant number of rounds (between 2 - 6) and at most $O(\sqrt{k})$ multiplications (but likely $O(1)$). In any case, the run-time of the last HybridGT call would be significantly better. In addition, the other sub-protocols

in BitGTA, namely – BitsEQ and Prefix OR – use more efficient building blocks that further improve efficiency. A more comprehensive comparison of the results was shown earlier in Table 2.

8.5 Summary of Comparison Protocols

The protocols presented in this section are all versions of bit-wise comparison of a public value c and some bit-wise shared $[r]$ (which is pre-processed). The general case of comparing two secret-shared values is easily reduced to this problem. The general technique is presented in Protocol 20, and costs only one additional round and invocation. Note that BitGT could be replaced by any of the protocols above. In practice, we use HybridBitGT here in order to do the selection based on the bit-length.

Algorithm 20: GT$([x], [y], k)$

$([r], [r]_B) \leftarrow RandNB(k+1, k)$;
$[a] \leftarrow [x] - [y]$;
$c \leftarrow Open(2^k + [a] + [r])$;
$c \leftarrow c \ mod \ 2^k$;
$[u] \leftarrow 1 - BitGT(c, [r]_B, k)$;
$[r'] \leftarrow \sum_{i=1}^{k} 2^i [r_i]_B$;
$[a'] \leftarrow c - [r'] + 2^k[u]$;
$[b] \leftarrow ([a] - [a'])(2^{-k} \ mod \ p)$;
return $[b]$

Finally, note that the greater-than building block can be locally transformed to any inequality relation. For example, to compute lower-than we simply need to compute $1 - GT$.

9 Improved Building Blocks

As mentioned earlier, the most important protocol to optimize over the integers is secure comparison, as many other protocols use it internally. With this understanding in mind, Section 8 was dedicated for this task specifically. In this section we introduce

generic building blocks that are derived from secure comparison and are used in more complicated arithmetic and math functions over the integers.

As before, each of our protocols will be accompanied by a correctness, security and complexity analysis, and all protocols are at least statistically-secure.

Table 3 summarizes the complexity of protocols in this section compared to previous results, which were best (in practice) for 32/64-bit integers

Protocol	This work		Previous results [29] [30]	
	Rounds	Communication	Rounds	Communication
Trunc	2	2	3	$k+2$
Mod2m	2	2	3	$k+2$
Mod/Int Division	3	6	6	$2k+4$
BitDec	2	$k+1$	4	$4k+1$

Table 3: Online complexity of secure integer computation protocols (assuming 32/64-bit integers)

9.1 Trunc and Mod

Shifting a secret shared integer to the left can be done locally by multiplying by 2^m, where m is the number of bits to shift. Right shift is equivalent to truncating m bits (i.e., multiplying by 2^{-m}). We have already shown a statistically accurate protocol in 1 round and 1 invocation. In fact, we have also presented the exact truncation protocol as well, as it is exactly the same as the secure comparison algorithm (Protocol 20), except that it allows specifying the value m, whereas secure comparison truncates all bits except the last. To show this, we specify here two protocols, one is Trunc (Protocol 22) and the other is Mod2m (Protocol 21). With these two in place, we can rewrite the secure comparison protocol as shown in Protocol 23. This should illustrate that all three problems are essentially the same (and all reduce to bit-wise comparison as discussed earlier). For these reasons, the correctness, security and complexity analysis are equivalent.

Algorithm 21: Mod2m$([x], k, m)$

$([r], [r]_B) \leftarrow RandNB(k+1, m)$;

$c \leftarrow Open(2^k + [x] + [r])$;

$c \leftarrow c \ mod \ 2^m$;

$[u] \leftarrow 1 - BitGT(c, [r]_B, k)$;

$[r'] \leftarrow \sum_{i=0}^{m-1} 2^i [r_i]_B$;

$[x'] \leftarrow c - [r'] + 2^m[u]$;

return $[x']$

Algorithm 22: Trunc$([x], k, m)$

$[x'] \leftarrow Mod2m([x], k, m)$;

$[b] \leftarrow ([x] - [x'])(2^{-m} \ mod \ p)$;

return $[b]$

Algorithm 23: GT$([x], [y], k)$

$[a] \leftarrow [x] - [y]$;

$[b] \leftarrow Trunc([a], k, k)$;

return $[b]$

General Modulo reduction - V1

For the general case, a slightly different protocol for modulo reduction is presented. The protocol requires two secure comparison calls (one is bit-wise, the other is over the integers) and is taken from [29]. However, the main overhead stems from the comparisons, which are constructed from the more efficient versions presented earlier.

Let a be the public modulus. Observe that Mod2m is more efficient for $a = 2^m$ because it is easy to generate a sharing of an integer $[r]$, taken from some large domain, and its shared bits. Then, in the online phase, we use $[r]$ to statistically mask the input, but only take the first m bits in order to do the reduction. This is equivalent to computing $[r \ mod \ 2^m]$ without cost. For a generic $2^{m-1} < a < 2^m$, computing $[r \ mod \ a]$ is not cost-less and requires the additional step of determining $[r \ mod \ a] = [r \ mod \ 2^m] - a[v]$, where $[v] = [r \geq a]$. We then set $[r'] = [r] - a[v]$, and

continue as Mod2m would with $[r']$ instead of $[r]$ and a instead of 2^m. Since $[r']$ is only given as an integer, the comparison is also done over the integers. Correctness and security follows from the same arguments and complexity compared to Mod2m costs an additional call to an integer comparison protocol.

Algorithm 24: ModV1$([x], k, a)$

$m \leftarrow \lceil \log(y) \rceil$;
$([r], [r]_B) \leftarrow RandNB(k+1, m)$;
$c \leftarrow Open(2^k + [x] + [r])$;
$c \leftarrow c \ mod \ a$;
$[v] \leftarrow BitGT([r]_B, a, k)$;
$[r'] = \sum_{i=0}^{m-1} 2^i [r_i]_B - a[v]$;
$[u] \leftarrow 1 - GT(c, [r'], k)$;
$[x'] \leftarrow c - [r'] + a[u]$;
return $[x']$

General Modulo reduction - V2

Given a general modulus $2^{m-1} < a < 2^m$, notice that the extra step above for determining $[v] = [r \geq a]$ is trivially zero if the m-th bit of r is also zero. In that case, we could simply call a variant of Mod2m that works for any a. Protocol 25 (ModP) shows this slightly modified version of Mod2m. Note that in the general case, the probability of ModP being correct is $\frac{1}{2} < \frac{a}{2^m} < 1$.

Algorithm 25: ModP$(c, [r]_B, k, a)$

$m \leftarrow \lceil \log(y) \rceil$;
$[u] \leftarrow 1 - BitGT(c, [r]_B, k)$;
$[r'] \leftarrow \sum_{i=0}^{m-1} 2^i [r_i]_B$;
$[x'] \leftarrow c - [r'] + a[u]$;
return $[x']$

A naive solution is to reveal a masked $[x]$ twice, once with $[r^{(1)}]$ and the second time with $[r^{(2)}]$, having $[r^{(2)}_{m-1}] = 1 - [r^{(1)}_{m-1}]$. In other words, at least one of the random values is going to lead to a correct result when using ModP. We can then

obliviously select the correct outcome by taking the inner product of the two results with the vector $([r^{(1)}_{m-1}], [r^{(2)}_{m-1}])$. Unfortunately, this solution is insecure and could potentially leak some information through the m-th bit of the revealed values, namely – $c^{(1)}_{m-1}$ and $c^{(2)}_{m-1}$. To see why this is true, recall that $c_i = x_i \oplus r_i \oplus carry_i = (x_i + r_i + carry_i)\ mod\ 2$. We then have $c^{(1)}_{m-1} = (x_{m-1} + r^{(1)}_{m-1} + carry^{(1)}_{m-1})\ mod\ 2$ and $c^{(2)}_{m-1} = (x_{m-1} + 1 - r^{(1)}_{m-1} + carry^{(2)}_{m-1})\ mod\ 2$. By summing the two together locally, an adversary can obtain $c^{(1)}_{m-1} + c^{(2)}_{m-1} = (2x_{m-1} + carry^{(1)}_{m-1} + carry^{(2)}_{m-1} + 1)\ mod\ 2 = (carry^{(1)}_{m-1} + carry^{(2)}_{m-1} + 1)\ mod\ 2$. Unless the carries are equal, some information may be leaked through these bits.

This can be corrected by first creating two k-bit random numbers $r^{(1)}$ and $r^{(2)}$ uniformly and independently at random. Then, 'correct' $r^{(2)}$ as follows:

$$\forall\ i \in \{0, ..., k\}\ r^{(2)}_i = \begin{cases} r^{(1)}_i & i < m-1 \\ 1 - r^{(1)}_i & i = m-1 \\ r^{(2)}_i & i > m-1 \end{cases} \tag{10}$$

In other words, we set the second generated random integer used to mask the input to be equal to the first one until the m-th bit. We then set the m-th bit of $r^{(2)}$ to be the additive inverse of the same bit in $r^{(1)}$ as before. Then, in order to prevent leakage in subsequent bits, we ensure that the rest of the bits of the two random integers are independently generated. Otherwise, the adversary could leak x_{m-1} by XORing (or subtracting) $c^{(1)}_m \oplus c^{(2)}_m$. The result would be the XOR of the carries, which would be 0 if no carry was set in both cases, implying $x_{m-1} = 0$ or $x_{m-1} = 1$ otherwise. By using different bits after the m-th bit, we ensure that no information leaks.

This analysis could be seen as using two one-time-pads for each bit. As long as the bits are the same, it is secure to use the same one-time-pad. After the deterministic step, the bits could be different, so we use a different pad for each. Protocol 26 shows the full solution.

Algorithm 26: ModV2($[x], k, a$)

$m \leftarrow \lceil \log(y) \rceil$;

$([r^{(1)}], [r^{(1)}]_B) \leftarrow RandNB(k+1, m)$;

$([r^{(2)}], [r^{(2)}]_B) \leftarrow RandNB(k+1, m)$;

for *i:=0,...,m* $-$ 2 **do**

  $[r_i^{(2)}] \leftarrow [r^{(1)}]_i$;

$[r_{m-1}^{(2)}] \leftarrow 1 - [r_{m-1}^{(1)}]$;

$c^{(1)} \leftarrow Open(2^k + [x] + [r^{(1)}])$; `// 1 rnd, 1 inv`

$c^{(2)} \leftarrow Open(2^k + [x] + [r^{(2)}])$; `// 1 inv`

$c^{(1)} \leftarrow c^{(1)} \ mod \ a$; $c^{(2)} \leftarrow c^{(2)} \ mod \ a$;

$[x'^{(1)}] \leftarrow ModP(c^{(1)}, [r^{(1)}]_B, k, a)$; `// See Protocol 13`

$[x'^{(2)}] \leftarrow ModP(c^{(2)}, [r^{(2)}]_B, k, a)$; `// In parallel`

$[x'] \leftarrow (1 - [r_{m-1}^{(1)}])[x'^{(1)}] + (1 - [r_{m-1}^{(2)}])[x'^{(2)}]$; `// 1 rnd, 2 inv`

return $[x']$

Correctness. In the pre-processing step, two random integers are generated according to Equation 10. Subsequently, two reveals publicly open $c^{(1)}$ and $c^{(2)}$. From the analysis above, we know that one of these is guaranteed to provide the correct result when sent to ModP, and that this occurs when $[r_{m-1}^{(j)}] = 0$ (for $j \in \{1, 2\}$). The last line obliviously selects the correct answer.

Security. Security follows from the security of ModP and the openings with the pre-determined randomness, which were discussed in depth above.

Complexity. The complexity of this protocol improves upon the previous version, as it avoids an expensive sequential integer comparison protocol. Instead, the cost is the same as two Mod2m calls in parallel, with an additional oblivious select that costs 1 round and 2 invocations. The total complexity depends on the underlying secure comparison scheme. For up to 64-bit integers, this amounts to 3 rounds and 6 invocations.

Integer division with a public divisor

For general-purpose division, the protocols in Section 10 should be used. However, a more efficient direct approach can be constructed for integer division with a public divisor. This is defined as $[q] = \lfloor \frac{[a]}{b} \rfloor$ and $[s] = [a] \; mod \; b$, which can be obtained using the general protocol ModV2 and truncation. The cost is 3 rounds and 6 invocations for up to 64-bit integers.

9.2 Bit Decomposition

Bit decomposition is an important building block in computation in general. In secure computation specifically, bit-decomposition was initially introduced in order to derive the first constant round protocols for secure comparison, as well as several other elementary operations over the integers [25]. Following the work of [26], the roles were flipped and secure comparison became the de-facto building block for deriving protocols over the integers (and later, approximations over the reals). Still, bit-decomposition is an important protocol for several common instructions, most notably bit-wise operations (e.g., OR, AND, XOR).

The first bit-decomposition with constant-rounds and $O(k \log k)$ communication was developed in [25], and improved by a constant factor in [26] (both having perfect security). While introducing the Postfix Comparison Problem (PFC), Toft introduced a constant-round (non-negligible constant), almost linear communication solution ($O(k \log^* k)$) [31], that was later reduced to linear for the statistically-secure case [30].

The reduction of bit decomposition to PFC has shown how a series (naively, $O(k)$) of secure comparisons could bit-decompose an integer. In previously proposed solutions, each comparison required $O(k)$ operations, so a naive solution would have had $O(k^2)$ complexity. Exploiting the prefix nature of PFC, Tord et al. [30] were able to approximate the solution using a single PrefixMul operation ($O(k)$ cost) along with $O(k)$ additional multiplications and reveals in parallel.

Below we introduce two protocols with what is believed to be a nearly optimal

online complexity, in the sense that the amount of rounds is exactly 2, and the amount of invocations are exactly $k + 1$. Given that the output has k secret elements, it is likely that achieving sub-linear results are not possible. Regardless, these protocols improve on the best known results by lowering the hidden constants in the cost of their protocols.

9.3 Naive BitDec

For $k \leq 64$, a naive solution presents a tight 2 rounds and $k+1$ multiplications which is better in practice than the previously best solution. The protocol is omitted for brevity, as it is trivial, and only a short explanation is given: the protocol starts by revealing a masked value $c \leftarrow x + r$. Then, BitGTS is called in parallel to compute $c \ mod \ 2^i > r \ mod \ 2^i$ for all k bits. Then, all that is left is to locally extract the decomposition from the series of comparisons as is shown in BitDec2 below.

The main shortcoming of this approach is the amount of pre-processing. For each input, we expand the required pre-processing by a factor of k. This leads to a computation of $k \cdot m$ look-up tables for each bit-decomposition.

9.4 BitDec from small comparisons

As an alternative, we would like to find a solution with similar performance in the online phase for $k \leq 64$, but without the associated offline overhead of doing k comparisons with look-up tables. We could use the same technique as before for converting a prefix problem into a comparison of small integers.

This solution is based on the PFC problem, which states that given $[x]$, a secret to be decomposed, it is sufficient to compute:

$$[x \ mod \ 2^i] \leftarrow [c \ mod \ 2^i] - [r \ mod \ 2^i] + 2^i([r \ mod \ 2^i > c \ mod \ 2^i]_B), \qquad (11)$$

where $c = x + r$ for a random r that statistically hides x. From this, it is trivial to locally compute the bit-decomposition of x using –

$$[x_i] \leftarrow ([x \bmod 2^{i+1}] - [x \bmod 2^i])2^{-i}. \tag{12}$$

The full protocol is shown in Protocol 27 below.

Algorithm 27: BitDec2($[x], k, m$)

$\beta \leftarrow \lceil \frac{k}{m} \rceil$;
$([r], [r]_B) \leftarrow RandNB(k, k)$;
$c \leftarrow Open(2^{k+1} + [x] + [r])$; `// 1 rnd, 1 inv`
$c \leftarrow c \bmod 2^{k+1}$;
$(c_{\beta-1}, ..., c_0) \leftarrow Blocks(c_B, m)$;
$([r]_{\beta-1}, ..., [r]_0) \leftarrow Blocks([r]_B, m)$;
$[d_0] = 0$;
for *i:=1,...,*β **do**
  $[d_i] \leftarrow BitGTS(c \bmod 2^{m \cdot i}, [r \bmod 2^{m \cdot i}])$;
for *i:=1,...,*$k+1$ *(in parallel)* **do**
  $i_B \leftarrow \min(\lfloor \frac{i}{m} \rfloor, \beta - 1)$;
  $\tilde{c} \leftarrow \sum_{j=0}^{i-i_B \cdot m} 2^j c_{j,i_B}$;
  $[\tilde{r}] \leftarrow \sum_{j=0}^{i-i_B \cdot m} 2^j [r_j]_{i_B}$;
  $[e_i] \leftarrow 2(\tilde{c} - [\tilde{r}]) + [d_{i_B}]$;
  $[u_i] \leftarrow 1 - CMP_k([e_i])$; `// 1 rnd,` k `inv`
  $[x \bmod 2^i] \leftarrow c \bmod 2^i - [r \bmod 2^i] + 2^i [u_i]$;
$[x_0] = [x \bmod 2]$;
for *i:=1,...,*$k-1$ **do**
  $[x_i] \leftarrow ([x \bmod 2^i] - [x \bmod 2^{i-1}])2^{-i}$;
return $[x]_B$

Correctness. The main idea is to reduce the comparison of each $[r \bmod 2^i > c \bmod 2^i]_B$, which is done over $O(k)$-bit integers, to that of approximately m-bit integers which we can solve efficiently (online and offline) using CMP_k. This only holds assuming as before that $2^m \sim k$. After publicly opening c, the parties split both c_B and $[r]_B$ to m-bit blocks. Note that for simplicity, the bits and blocks are

numbered starting from 0. The first interesting part of the protocol is when all the parties compute the values of $[d]$. Each $[d_i]$ contains the result of $c \ mod \ 2^{m\cdot(i-1)} \leq r \ mod \ 2^{m\cdot(i-1)}$, namely – the comparison outcomes are sampled in m intervals, and $[d_i] \in \{-1, 0, 1\}$ stores for all bits in a block i, the comparison result up to this block.

The second loop occurs in parallel for each bit separately. i_B is a pointer to the current block; $\tilde{c}$ and $[\tilde{r}]$ represent m-bit integers constructed from all bits in this block. For example, for $i = 11$ and given $m = 8$, $\tilde{c}$ is comprised of bits $8, ..., 11$ of c. This is the same as computing $\tilde{c} \leftarrow trunc(c \ mod \ 2^i, i_B)$. The case for $[r]$ is equivalent.

Observe that for every i, the comparison of $c \ mod \ 2^i$ and $r \ mod \ 2^i$ is determined by the boolean circuit:

$$([\tilde{r} \neq \tilde{c}]_B \wedge [\tilde{r} \leq \tilde{c}]_B) \vee ([\tilde{r} = \tilde{c}]_B \wedge [r \ mod \ 2^{m\cdot(i_B-1)} \leq c \ mod \ 2^{m\cdot(i_B-1)}]_B). \qquad (13)$$

However, this is a boolean of depth 2 with two inequalities and one equality test. We can reduce it to a single round by shifting $\tilde{c} - [\tilde{r}]$ by one bit (i.e., multiplying by two) and adding $[d_{i_B}] \in \{-1, 0, 1\}$ to the result. Cached in $[e_i]$ for each bit, we then use the bounded small integers comparison (CMP_k) and invert the resulting bit in order to obtain the needed $[u_i] \leftarrow [r \ mod \ 2^i > c \ mod \ 2^i]_B$. Finally, using Equations 11 and 12, the parties locally compute the bit decomposed $[x]$.

Security. Other than invocation of secure sub-protocols (BitGTS, CMP_k), all operations are local except for one public reveal. As $[r]$ statistically hides $[x]$, no information is learned except with negligible probability.

Complexity. The online complexity of this protocol is exactly the same as that of the naive version – 2 rounds and k+1 invocations. However, there is a factor m reduction in the amount of pre-processed tables required. For 32-bit and 64-bit integers, only 4, 8 (respectively) tables are used. Since we have already established that it is not likely that a constant-round protocol could be implemented with less than $O(k)$ communication (since that is also the size of the output), we conclude that this protocol has (nearly) optimal complexity in the online phase. Also, this protocol

achieves the same online complexity even in the asymptotic case, making it more efficient than the previously best protocol in [30]. However, in this case the amount of pre-processed lookup tables are again $O(k)$.

10 Constant Rounds, Sub-linear Secure Division

Secure division is the most complex of the elementary protocols [28]. Many of the previously introduced building-blocks are invoked as sub-protocols, most notably Bit Decomposition and Prefix OR, both of which are believed to have a linear communication lower bound (for constant rounds protocols). It therefore appears contradictory to be able to construct a constant-round solution that still has sub-linear communication complexity, and indeed all current solutions involve a log-rounds and $O(k \log k)$ communication complexity (e.g., [54], [28]).

In this section, we present the first solution for a constant rounds, sub-linear communication complexity division protocol that works for integers and fixed-point representations using a LSSS. This solution is only probabilistically correct, but the probability of failure is negligible (considering k is large enough or a parameter l is set accordingly). The solution involves two parts. First, achieving sub-linear complexity requires improving the algorithm for normalizing a secret integer to the range $(0.5, 1]$. This operation has notoriously been the largest performance bottle-neck in secure division and reciprocal operations. We present an asymptotically improved protocol in section 10.2 and a practically improved one ($k \leq 64$) in 10.3.

Second, multiplicative division requires $O(\log_2 k)$ rounds to reach a certain precision. Since until this work, the initial normalization was overwhelmingly less efficient than the actual division procedure, there was less need to reduce the number of rounds. However, given an efficient normalization algorithm, we also provide a constant rounds solution at the cost of two rounds operating in a (significantly) larger field. This is shown in Section 10.4.

The results of this section are summarized in Table 4 for the 32/64-bit case, and separately for the asymptotic case. The first settles for linear communication com-

plexity but attempts to reduce constants (most notably – the number of rounds), while the latter achieves sub-linear communication complexity. Note that the protocols for Norm and Division can generalize to other secure computation protocols in the fixed-point or floating-point representation, as alluded to in Section 10.5.

Source	Norm		FPDiv/FPReciprocal	
	Rounds	Comm.	Rounds	Comm.
[29]	$2\log k + 2$	$1.5k \log k + k + 2$	$2\log k + 4\theta + 7$	$1.5k \log k + k + 4\theta + 7$
Proposed	4	$2k + 2$	11	$2k + 3\theta + 7$
Proposed (asym.)	$O(1)$	$O(\sqrt{k})$	$O(1)$	$O(\sqrt{k} + \theta)$

Table 4: Online complexity of improved secure fixed-point computation protocols

10.1 Multiplicative (Iterative) Division

Unlike digit recurrence approaches which converge slowly, multiplicative division schemes have quadratic convergence, enabling a division circuit to take a logarithmic number of iterations in the required precision (compared to linear in digit recurrence). The two most common schemes are based on Newton-Raphson method or series expansion. In particular, the latter is usually based on Goldschmidt's method, which we focus on in this work. Both methods are functionally similar, with the exception that Goldschmidt's method has two independent multiplications occurring in each iteration, whereas Newton-Raphson method requires two sequential multiplications, thus doubling the amount of needed rounds. In addition, as we will later show, it is possible to collapse the logarithmic circuit into a single (or small constant) round with higher communication complexity.

Let a, b be the dividend and divisor respectively. Multiplicative division starts by finding an initial approximation of the reciprocal of the divisor. To ensure fast convergence, b is first normalized to the range $\hat{b} \in (0.5, 1]$, as shown in sections 10.3 and 10.4. Then, using the normalized result we can find a first order approximation: $\hat{y_0} \approx \frac{1}{\hat{b}} = c_0 - c_1\hat{b}$. Finding the optimal coefficients [72] and scaling back the estimated reciprocal yields:

$$y_0 = 2.928 - 2b, \tag{14}$$

with error –

$$\epsilon_0 = 1 - by_0. \tag{15}$$

Empirically assigning different values for b in Equation 15 (this is easier if we test values of the normalized $\hat{b} \in (0.5, 1]$ instead), we can see that the initial approximation has a bounded error of $|\epsilon_0| \leq 0.072$ providing nearly 4-bits of precision (≈ 3.8).

After the initial approximation, Goldschmidt method operates by multiplying both a, b by successively better approximations y_i of the reciprocal, until the denominator converges to 1 and the nominator contains the approximated quotient.

$$\frac{a}{b} = \frac{ay_0y_1...y_{\theta-1}}{by_0y_1...y_{\theta-1}} \leftrightarrow \frac{a_{i+1}}{b_{i+1}} = \frac{ay_i}{by_i} \tag{16}$$

At this point, subsequent y_i's need to be determined (currently, we have only obtained the initial approximation), with the goal of driving the denominator to 1. Observe that already after the first iteration, the denominator is equal to $b_1 = by_0 = 1 - \epsilon_0$ (from Equation 15). Since the absolute value of the initial error ϵ_0 is strictly lower than 1, then b_1 is close to 1 as well. This is why the initial approximation is important for fast convergence. In order to obtain quadratic convergence, it is sufficient to set each y_i using the following recursive rule:

$$y_i = 2 - b_i = 2 - b_{i-1}y_{i-1} = 1 + (1 - b_{i-1}y_{i-1}) = 1 + \epsilon_i. \tag{17}$$

Equation 17 also implies that $b_i = 1 - \epsilon_i$. Using these relations for y_i and b_i, we can show by induction that $b_{i+1} = b_iy_i = (1 - \epsilon^{2^i})(1 + \epsilon^{2^i}) = 1 - \epsilon^{2^{i+1}}$. Therefore, in each iteration, the approximation's accuracy is double that of the previous estimate, as desired. The number of iterations (and the depth of the circuit in the naive approach) is set to:

$$\theta = \log\lceil\frac{k}{c}\rceil, \tag{18}$$

where k is the desired target precision in bits and c is the precision of the initial approximation.

10.2 Norm

While multiplicative division algorithms have quadratic convergence, they require an initial normalization step, which scales the divisor to the range $(0.5, 1]$. Since the divisor is a secret, doing so securely is not trivial and highly inefficient. In fact, as mentioned earlier this is the main bottleneck in secure division protocols, taking significantly more rounds and invocations compared to the actual iterative division process.

Previous solutions (e.g., [28]) were mainly expensive due to their log-rounds implementation of Bit Decomposition and Prefix OR operations. We have already shown how to achieve these in 1-2 rounds and $\approx k$ invocations (without any hidden constants). With these improvements, we can already obtain a very low constant-rounds solution for normalization. This is the basis of NormS presented below, for practical applications using common bit-lengths (e.g., up to 64-bits). However, in the asymptotic case, this still implies the normalization, and therefore – division, reciprocal, as well as other protocols such as square root and converting to floating point representation, all asymptotically require $O(k)$ communication. By showing a solution that is still constant-rounds but requires sub-linear communication complexity, we immediately improve all of these as well (for the asymptotic case).

We now describe the main developments leading to this protocol. Given a secret $2^{m-1} \leq |x| \leq 2^m$ $(m \leq k)$, the goal is to find a normalizing $[v] \leftarrow 2^{k-m}$ and use it to obtain a normalized value $[u] \leftarrow [x][v] \in [0.5, 1)$, with between $k-1$ and k bits of precision. In other words, Norm normalizes a secret input such that it is between 0.5 and 1, and returns the normalizing exponent as well. The first step in obtaining a sub-linear communication protocol is similar to the one used for sub-linear secure

comparison [27]. First, the parties publicly open a masked $[x]$ and store it as c. Both c and the pre-processed randomness used as the mask are split to $\sqrt{k}$-bit blocks. These undergo $\sqrt{k}$ equality tests in parallel, followed by a Prefix OR call that yields a result marking which blocks of c and $[r]$ are equal and which are not.

From this point and on, the sub-linear comparison protocol and the sub-linear normalization protocol differ significantly. In the secure comparison case, the parties had enough information at this point to locally compute the first block in which c and $[r]$ differ (and for comparison that is all we need). For normalization, things are more involved, as we are not interested in c and $[r]$ directly, but rather the underlying $[x]$. Specifically, our goal is to accurately locate and obliviously select the MSB of $[x]$. This is trivial to do using calls to BitDec and Prefix OR, but both of these are believed to have a linear lower bound (for constant rounds protocols).

To solve this, the idea is to identify the last $\sqrt{k}$-bit block where $[x]$ is not all zeros. Naively, attempting to look at the last bit c_i that differs from r_i could prove to be a false-positive. To see why this is true, first observe that the following relationship holds for c:

$$c = x + r = x \oplus r \oplus cin. \tag{19}$$

The interpretation that follows is that for each bit i, if $r_i \neq c_i$ it is either because cin_i is set or x_i is set, but we do not know which, and directly computing the carries is an in-efficient procedure [25]. When $c_i = r_i$, then both $x_i = 0$ and $cin_i = 0$.

Therefore, if we look at the last differing bit (LDB), we have no way of telling whether this was due to x_i being set (true positive) or $cin_i = 1$ (false positive). Instead, if we look at the LDB i, and the series of m bits preceding it and then ask a similar question – what is the probability that the LDB has $x_i = 0$ and *all* m bits preceding the LDB have $x_j = 0$ as well? Observe that this can only occur if all $r_j = 1$. Otherwise, there exists some $x_j \neq 0$ in contradiction to the requirement. Since each bit of r is selected independently and uniformly at random, we have that –

$$Pr[\wedge_j\{x_j = 0 \mid j \in \{i-m, ..., i-1\} \wedge \ i := LDB\}] =$$
$$Pr[\wedge_j r_j = 0] = \prod_{j=1}^{m} Pr[r_j = 0] = 2^{-m}$$

Thus, the probability of this occurring is 2^{-m}, which is negligibly small for large enough m. Note that as our protocol selects $\sqrt{k}$ blocks, we need to slightly modify the above equation. In the case of blocks, we are interested in reducing the problem of normalizing from a k-bit number to a $\sqrt{k}$ integer. Using the secure comparison idea, we already know how to obliviously select the block where the LDB resides. In the worst-case, the LDB is also the first bit of this block and we do not know if that is a false positive or not. But using the idea above, we can select the block preceding it as well and be confident that even if the LDB is indeed a false-positive, then w.h.p the left-most $x_j = 1$ is in the preceding block. Given that this block has $\sqrt{k}$ bits, we conclude that the probability that we miss the most significant bit where x is set is at most $2^{-\sqrt{k}}$. If k is sufficiently large then the probability of error is negligible. Otherwise, we could use a parameter l to control the amount of blocks we select and examine, to make the probability arbitrarily small. Specifically, let l be the number of preceding blocks we select, then the probability of error is at most $2^{-l\sqrt{k}}$.

Algorithm 28: Norm$([x], k, l)$

$([r], [r]_B) \leftarrow RandNB(k, k)$;

$c \leftarrow Open(2^{k+1} + [x] + [r])$; // 1 rnd, 1 inv

$c \leftarrow c \bmod 2^{k+1}$;

$(c_{\sqrt{k}-1}, ..., c_0) \leftarrow Blocks(c_B, \sqrt{k})$;

$([r]_{\sqrt{k}-1}, ..., [r]_0) \leftarrow Blocks([r]_B, \sqrt{k})$;

for *i:=0,...,*$\sqrt{k} - 1$ **do**

  $[d_i] \leftarrow 1 - BitsEQ(c_i, [r]_i)$; // 1 rnd, $\sqrt{k}$ inv

$[e]_B \leftarrow PreOR([d]_B)$; // 1 rnd, $\sqrt{k}$ inv

for *i:=0,...,l (in parallel)* **do**

  $[f]_B \leftarrow BitTrunc([e]_B, i)$;

  $[x \bmod s_i], [s_i] \leftarrow BitMod2s((c_{\sqrt{k}-1}, ..., c_0), ([r]_{\sqrt{k}-1}, ..., [r]_0), [f]_B, \sqrt{k})$;

  // $O(1)$ rnd, $O(\sqrt{k})$ inv

  $[x'_i] \leftarrow ([x] - [x \bmod s_i])[s_i]^{-1}$; // $l+1$ rnd, $l+1$ inv

  $[g_i] \leftarrow 1 - EQZ([x'_i])$; // 2 rnd, 2 inv

$[h']_B \leftarrow PreOR([g]_B)$; // 1 rnd, $l+1$ inv

for *i:=0,...,l* $- 1$ **do**

  $[h_i] \leftarrow [h'_i] - [h'_{i+1}]$;

$[h_l] \leftarrow [h'_l]$;

$[\tilde{x}] \leftarrow \sum_{i=0}^{l} [h_i][x'_i]$; // 1 rnd, $l+1$ inv

$[\tilde{s}] \leftarrow \sum_{i=0}^{l} [h_i][s_i]$; // $l+1$ inv

$[\tilde{x}]_B \leftarrow BitDec([\tilde{x}], \sqrt{k})$; // 2 rnd, $\sqrt{k}+1$ inv

$[y']_B \leftarrow PreOR([\tilde{x}]_B)$; // 1 rnd, $\sqrt{k}$ inv

for *i:=0,...,*$\sqrt{k} - 1$ **do**

  $[y_i] \leftarrow [y'_i] - [y'_{i+1}]$;

$[y_{\sqrt{k}-1}] \leftarrow [y'_{\sqrt{k}-1}]$;

$[t] \leftarrow \sum_{i=0}^{\sqrt{k}-1} 2^i [y_i]$;

$[v] \leftarrow [\tilde{s}][t]^{-1}$; // 1 rnd, 3 inv

$[u] \leftarrow [x][v]$; // 1 rnd, 1 inv

return $([u], [v])$

Correctness. Norm describes the procedure explained above and correctness follows. Note that for simplicity, we assume k is square (otherwise, it can be padded with zeros). We also consider $[x]$ to be unsigned in this protocol and the next. The sign could be extracted efficiently using the secure comparison protocols produced earlier, as shown in [28].

Security. Security follows from the fact that only a single value is publicly revealed in the outer protocol. The rest are revealed in sub-protocols which are secure. The reveal in the beginning is exactly the same as in other protocols and uses a mask $[r]$ that statistically hides the input $[x]$. We conclude that Norm is secure and provides statistical security.

Complexity. All operations have a constant number of rounds and at most $O(\sqrt{k})$ invocations.

Finally, this protocol proves by construction the following theorem.

Theorem 2. *Normalization of a secret input can be computed securely in $O(1)$ rounds and $O(\sqrt{k})$ invocations, with only negligibly small (in $\sqrt{k}$) probability of obtaining an incorrect result.*

10.3 NormS

While we have proved better bounds for both round and communication complexity, in practice a constant-round with linear communication variation of the log-depth protocol in [28] would be preferable for most applications. While asymptotically less efficient, it involves less rounds, which is more important when k is small. The reasoning is similar to that shown in Section 8, where secure comparison in practice could be significantly optimized when considering practical values of k.

Algorithm 29: NormS($[x], k$)

$[x]_B \leftarrow BitDec2([x], k)$; // 2 rnd, $k+1$ inv

$[y']_B \leftarrow PreOR([x]_B)$; // 1 rnd, k inv

for *i:=0,...,*$k-1$ **do**

  $[y_i] \leftarrow [y'_i] - [y'_{i+1}]$;

$[y_{k-1}] \leftarrow [y'_{k-1}]$;

$[v] \leftarrow \sum_{i=0}^{k-1} 2^{k-i-1}[y_i]$;

$[u] \leftarrow [x][v]$; // 1 rnd, 1 inv

return $([u], [v])$

Correctness. Note that NormS is equivalent to Norm, except that it does not go through the initial process of reducing the location of the most significant bit to a $\sqrt{k}$-bit block. It also computes $[v] = 2^{k-m}$ directly, where m is the location of the MSB, whereas the protocol above first computes the location of the block and then multiplies that by the location inside the block.

Security. In this case, no information is revealed in the outer protocol, which only invokes sub-protocols (multiplication, BitDec, PreOR) that are known to be secure. By composition, and the fact that at least one protcol has statistical security, NormS is also statistically secure.

Complexity. Since unlike the sub-linear protocol, BitDec and PreOR are computed against the entire secret k-bit value, the asymptotic complexity is linear. However, the number of rounds is significantly smaller – a total of 4. The exact invocation count is $2(k+1)$.

10.4 Division (Reciprocal)

All previous secure division protocols (e.g., [54], [28], [33]) used a log-depth circuit to implement multiplicative division. To achieve a constant-rounds protocol, observe that Equation 16 could be re-formulated into the following problem of finding the reciprocal:

$$\frac{1}{b} = y_0 y_1 ... y_{\theta-1} = y_0(1+\epsilon_0)(1+\epsilon_0^2)...(1+\epsilon_0^{2^{\theta-2}}). \tag{20}$$

Since ϵ_0 can be obtained in a single round from Equation 15, then assuming we can find a protocol for computing multiple powers of a secret $[x]$ in constant rounds, then the problem is solved as all that is left is a single round of unbounded fan-in multiplication of the terms. Such a protocol is presented in Protocol 30 below.

BatchExpFP

The goal of BatchExpFP is to compute several powers of a secret $[x]$ in parallel, and truncate them to the desired precision of the fixed-point representation. The last part is needed for our use case of truncating the excess bits resulting in Goldschmidt's method, and since our fixed-point representation has a public precision parameter f, we ensure that all results adhere to this standard. For the same reasons, it is likely to assume that our field $\mathbb{Z}_p$ is too small for computing large enough exponents of x, which is the reason a conversion to and from q is required, where $q > \max\{l_i\} \cdot f$. If p meets these requirements already then the conversion steps are omitted. Since we are working with two different fields in this protocol, we explicitly use a subscript for the shares to denote in which field they are stored in.

Algorithm 30: BatchExpFP$([x]_p, l, p, q, f)$

$([r]_q, [r^{-l_1}]_q, ..., [r^{-l_m}]_q) \leftarrow RandExpInv(q, l)$;

$[x]_q \leftarrow ConvertZp2Zq([x]_p, q)$; `// 1 rnd, 1 inv`

$c \leftarrow MulPub([x]_q[r]_q)$; `// 1 rnd, 1 inv`

for *i:=1,...,m(in parallel)* **do**

- $[y'_{l_i}]_q \leftarrow (c^{l_i} \ mod \ q)[r^{-l_i}]_q$;
- $[y_{l_i}]_q \leftarrow TruncPr([y'_{l_i}]_q, (l_i - 1)f)$; `// 1 rnd, m inv`
- $[y_{l_i}]_p \leftarrow ConvertZp2Zq([y_{l_i}]_q, p)$; `// 1 rnd, m inv`

return $([y_{l_1}]_p, ..., [y_{l_m}]_p)$

Correctness. BatchExpFP receives a secret $[x]_p$, where $x > 0$, a list l of public exponents to compute (the list has size m which is implicit), and two primes denoting the field of origin (p) and the target field (q) to do the computations in. The last input

f marks the desired fixed point precision/scaling factor. The offline phase involves generating a random secret shared value and the inverse of its powers. This is simply a batch version of the offline phase described in [25]. We assume that q and l are known apriori to enable the randomness generation to be done offline.

After the offline phase, the protocol starts by converting the secret $[x]_p$ into a large enough field $\mathbb{Z}_q$, where $q > \max\{l_i\} \cdot f$ as mentioned above. The check that ensures this is left implicit, particularly since q is public and known in advanced. Note that since the underlying x is positive, the conversion will be *correct* (i.e., $x \ mod \ p = x \ mod \ q$). After converting, the parties publicly reveal the result $c = (x \cdot r) \ mod \ q$. With this, they can locally compute each one of the exponents $l_i \in l$, since $y'_{l_i} \ mod \ q = (c^{l_i} r^{-l_i}) = ((x^{l_i} \cdot r^{l_i}) r^{-l_i}) \ mod \ q = x^{l_i} \ mod \ q$. Because q is large enough to accommodate $\max\{l_i\}$, then no overflow will occur. The last two steps involve truncating each exponent to f bits of precision and converting the result back into $\mathbb{Z}_p$.

Security. The only potential leakage occurs in the call to MulPub. Since $[r]_q$ is a random field element in $\mathbb{Z}_q$ then it perfectly masks $[x]_q$ and no information is leaked, except if $x \neq 0$, which we assume is not the case. If we cannot make this assumption, then we can condition the execution of this protocol based on the result of the equality test $[x = 0]$. This could be done obliviously as in [25]. All other sub-protocols have either perfect or statistical security and therefore – BatchExpFP is statistically secure as well.

Complexity. The protocol has a total of 4 rounds and $2m + 2$ invocations (in the online phase), where m is the length of the list l (for our use cases $m = \theta - 1 = O(\log_2 k)$). This is a result of an initial conversion of one field element, followed by a public reveal, and then m calls to TruncPr and m conversions of the outputs. If no conversions are needed, then the complexity is reduced to 2 rounds and $m + 1$ invocations. Finally, we note that for the integer case where no truncation or conversion is needed, batch exponentiation only costs a single round and invocation in the online phase.

FPDiv

We are now ready to present the full division protocol, shown in Protocol 31.

Algorithm 31: FPDiv$([a], [b], k, f)$

$\theta \leftarrow \lceil \log \frac{k}{3.75} \rceil$; $\alpha \leftarrow fld(2.928_k)$; $l \leftarrow \{2, 4, ..., 2^{\theta-1}\}$;

$q \leftarrow NextPrime(2^{2^{\theta-1}f} + 1)$;

$([\hat{b}], [v]) \leftarrow Norm([b], k)$; `// See Prot. 28 or 29`

$[\hat{y_0}] \leftarrow \alpha - 2[\hat{b}]$;

$[y_0] \leftarrow [\hat{y_0}][v]$; `// 1 rnd, 1 inv`

$[d] \leftarrow [b][y_0]$; `// 1 inv`

$[y_0] \leftarrow TruncPr([y_0], 2(k-f))$; `// 1 rnd, 1 inv`

$[d] \leftarrow TruncPr([d], f)$; `// 1 inv`

$[\epsilon_0] \leftarrow fld(1_f) - [d]$;

$([\epsilon_1], ..., [\epsilon_{\theta-1}]) \leftarrow BatchExpFP([\epsilon_0], l, p, q, f)$; `// 4 rnd, 2θ inv`

$[y] \leftarrow [a][y_0] \prod_{i=0}^{\theta-1} (fld(1_f) + [\epsilon_i])$; `// 1 rnd, θ + 2 inv`

$[y] \leftarrow TruncPr([y], (\theta+1)f)$; `// 1 rnd, 1 inv`

return $[y]$

Correctness. The protocol starts with initializing the needed parameters as described earlier: θ is the number of Goldschmidt iterations to compute (in parallel); α is used for the initial approximation of the reciprocal; l is the list of exponents to compute; and q is a large enough prime to compute the exponents without overflow in BatchExpFP. The protocol starts by normalizing the secret divisor $[b]$ and computing the first approximation $[y_0]$ (requires multiplying by $[v]$ to undo the normalization). The parties then compute $[\epsilon_0]$ from Equation 15, and compute the needed exponents for Equation 20 (i.e., the reformulation of Goldschmidt's method we use to flatten the circuit). This is done by a call to BatchExpFP presented earlier. Finally, Equation 20 is computed with only a single unbounded fan-in multiplication call. This requires $p > k + \theta f$, or could be avoided with an additional round or two at most. The last truncation ensures that the result has f bits of precision as required.

Security. As there are no elements revealed in the outer protocol, the protocol is (statistically) secure by the composition of secure sub-protocols.

Complexity. The exact number of rounds depends on which version of Norm is used. We use the version with less rounds here, as for practical bit-lengths reducing the number of rounds is likely to yield better performance than reducing communication. The exact number of rounds then amounts to 11 rounds and $2k + 3\theta + 7$ invocations.

Note on reciprocal. Observe that computing the reciprocal of a secret number $[x]$ can be done by setting a public $a = fld(1_f)$ and calling FPDiv internally, as shown in Protocol 32.

Algorithm 32: FPReciprocal$([x], k, f)$

$a \leftarrow fld(1_f)$;

$[y] \leftarrow FPDiv(a, [x], k, f)$;

return $[y]$

Finally, we can summarize the results in the theorem shown below.

Theorem 3. *Assuming some operations are computed in a target field with a prime $q > \max\{l_i\} \cdot f$, division and reciprocal operations can be computed securely in $O(1)$ rounds and $O(\sqrt{k})$ invocations, with only negligibly small (in $\sqrt{k}$) probability of obtaining an incorrect result.*

10.5 Other Applications

The techniques developed in this section have other important applications. As these are fairly trivial reductions from either secure normalization or division (or both), we omit the implementation details and focus on the implications.

Floating-point representation. Representing real numbers using floating-point representation allows for greater precision and a larger (dynamic) range compared to fixed-point representation. Essentially, for the same price (i.e., number of bits), we can represent larger values with higher precision. This works by having a dynamic gap between values – namely, smaller numbers in the range are closer to each other while larger numbers are farther apart. This is unlike fixed-point representation which has a uniform gap.

The cost of using floating-point representation is more complex arithmetic. Nowadays, modern processors have a built-in floating-point unit (FPU) for dealing with this efficiently, thus it has become the industry standard except for lower-end processors that are only equipped with arithmetic logic units (ALUs). In secure computation, the problem is compounded, and recent attempts to implement floating-point operations [33, 35, 34, 32] are almost conclusively less-efficient. This is mainly because floating-point addition, which is free for integers and fixed-point operations, requires interaction. For the same reason, protocols such as division/reciprocal that requires normalization, are more efficient.

The link between fixed-point and floating-point representation lies in the normalization. This is also the basis of converting the two schemes [33]. Since this operation is considered as the main bottleneck, our improvements immediately translate to more efficient conversion protocols in both directions, allowing hybrid protocols to be developed. This is the main idea in [35], except that they did not have an efficient normalization scheme and were therefore forced to start with a normalized floating-point representation and move to fixed-point representation (and back) as needed. Another hybrid approach used conversion to garbled circuits (which are more suited for bit-wise operations), but is still largely less efficient, as the authors reported that some elementary operations would take minutes to complete [32].

Instead, an approach starting from a fixed-point representation that converts to floating point when certain operations are called (e.g., division, square root, logarithm and exponent) would lead to better performance and still allow cost-less linear operations. Such an approach could be employed using our normalization protocols to replace the inefficient ones used in the conversion protocols of [33].

Square root. Computing the square-root could be done using Goldschmidt's method as well, and is largely the same as division [36]. With our efficient normalization and a constant-rounds solution (as opposed to a log-depth circuit) for computing series expansion, the square root version could be similarly adapted and improved.

Series approximations in general. Many other important math primitives over the reals could be approximated using a Taylor series or Chebyshev polyno-

mials. These include trigonometric functions, logarithm, exponent, as well as division/reciprocal and square root. For the latter, we have shown a quadratic convergence method which is faster. These approximations often require the inputs to be normalized to some range (e.g., needed for log), so our Norm protocols apply to these cases. Also for the series approximation itself, our solution to the iterative method is easily adapted. BatchExpFP can compute the polynomial terms (for a desired precision), followed by a summation of all terms (instead of a product).

11 Implementation

Building on the theoretical and practical improvements presented thus far, a reference implementation of the system was developed. An earlier version of this system appeared in [71].

11.1 The Network

On a low-level, all players in our system (owners, services and computing parties) are nodes in our peer-to-peer network. Each player also maintains a connection to the Bitcoin blockchain network.

The network protocol is implemented in Python using Twisted, which is an asynchronous, event-driven networking library. Contrast this with Bitcoin, which uses a synchronous model. For peer-discovery, we follow a similar protocol to that of Bitcoin (a succinct summary can be found in [42]). Our default client contains a configuration file with seed nodes to connect to. When a new node connects to another node, it receives a sample of known addresses, which it attempts to connect to. Nodes propagate new connections so that other nodes can update their list of known peers. This forms a random sparse graph with a small diameter, which is efficient for propagating information.

We also introduce an additional concept of a *federated node*, that did not exist in the theoretical model. The rationale behind this is that we needed to bridge the gap between the formal model of a blockchain and a specific practical implementation.

Specifically, there are two model assumptions that do not yet hold in practice: the blockchain has infinite storage; and it can run any computation (on public data) [6]. The federated node solves this by storing most of the public state, namely – commitments to inputs, transcripts of computations and some other meta-data such as information about the quorums. It also executes the parts of the smart contracts that bitcoin scripting does not support. We explain how this is done in detail and how it affects the security model in Section 11.4.

We stress that significant community efforts are currently dedicated to solving the scalability concern (e.g., [68]), so relying on some amount of federation in the system should be considered a temporary solution. In addition, while the federated node can disrupt the functionality of the network, it cannot break privacy of the data. Also note that we only rely on the blockchain for public consensus and incentives, so we are not required to use Bitcoin specifically [7].

11.2 Clients

The low level network protocol is wrapped into daemon. On top of that, we implemented the following clients, based on their role in the system:

1. **Computing Party.** This is where most of the interesting logic is. These are the parties who store secret data and execute computations when requested in return for rewards. We developed a *distributed virtual machine* (DVM) that interprets code in runtime as secure MPC protocols (see Section 11.3).

2. **Federated Node.** As mentioned, this node holds most of the public state and executes Contracts (while making the outcome visible on the blockchain). Since it only exists to assist the blockchain, it never holds any private information.

3. **Owner/Service.** This is daeamon wrapped into a thin client with a software library to interact with it. The library supports secret sharing and sending computations in the low level language of the DVM. The goal of the DVM

[6] Some blockchains already support this. For example, Ethereum – https://www.ethereum.org.
[7] Our choice of Bitcoin was due to its relative maturity.

is to be a generic intermediate representation (IR) which existing high-level languages can compile down to.

Computing parties and federated nodes persist their data in a local LevelDB[8] database.

11.3 Distributed VM (DVM)

The computing parties in our network collectively form a single decentralized computer. In practice, for each computation only a quorum is sampled, but we omit this detail here. To implement this idea, we developed a distributed virtual machine (DVM) for executing low-level code. Our approach is different from previous work on programming languages for secure computation (see [45], [47] and [46] for some recent work). Instead of developing a domain-specific language, which requires developers to learn new tools, we focus on the runtime environment. With a coherent specification of a DVM, it should be easy to develop thin software libraries that can compile *existing* high-level programming languages into bytecode that our network can execute in a distributed fashion. This is how, for example, JVM [9] can be used as the underlying runtime engine for languages other than Java, such as Python and Ruby [10].

Conceptually, a DVM differs from a local VM in two major aspects, that boil down to the VM being dependent on the *collective state* of several parties:

1. **Network awareness.** Since the DVM is a network of machines, we need special opcodes for network communication.

2. **Synchronization.** Non-linear operations cannot be computed locally and require one or more rounds of communication. Specifically, from the view of a single machine, we need a way to pause the local execution until some needed information is received from the network.

[8] http://leveldb.org/
[9] https://en.wikipedia.org/wiki/Java_virtual_machine
[10] https://en.wikipedia.org/wiki/List_of_JVM_languages

Our DVM implementation is modeled after CPython, the most commonly used runtime environment for Python. CPython is well documented and understood, so we will only provide necessary details here for our implementation. In general, any high-level python code block (e.g., a function, a module or a class) is compiled down to a *code object*. Loosely speaking, a code object is a set of instructions (i.e., a bytecode) coupled with lists of symbols such as variable names and constants (including code objects for nested code blocks).

To illustrate how the VM works, let us consider the execution of a single code object (e.g., a function). A code object is essentially an *intermediate representation* (IR) of high-level code, so the interpreter first needs to create the runtime equivalent of it – a *frame object*. This object includes, among other things, some random-access memory with different program-scopes (locals, globals and builtins). These are populated during initialization of the frame, which is determined at runtime. Nested code blocks, such as nested functions, create new frame objects and add them to a *call-stack*. This allows for a recursive evaluation of code and is common in VMs. Conversely, *clauses* such as loops do not create a separate frame and are managed within a single frame using block objects. Block objects are conceptually similar to frames and provide another level in the hierarchy of execution. As a result, each frame has its own block-stack. Finally, every frame is evaluated as a stack machine, with a value stack that interacts with the RAM.

Our additions to python's VM include several new opcodes for network operations (see Table 5 for details). To achieve synchronization, we extend the meaning of a block object to encapsulate at most one round of communication. This means that a single frame object will create a block object for every round of communication, or in other words – for every multiplication gate.

As an example, observe the compiled code for an addition gate and a multiplication gate in Table 6, which illustrates the advantages of having a dynamically typed IR language. First, while not seen in these examples, not all code is compiled into a large static circuit, but rather instructions are evaluated one-by-one and only when needed, a block is constructed for communication on-the-fly. Second, the dynamic

Opcode	Description
SEND_SHARE n	Pop id from TOS; Pop $n-1$ elements; Send them to id
SEND_RESULT_SHARE	Send output gate share
BC_SHARE n	Broadcast top n elements in the stack
RECONSTRUCT t	Pop t elements and reconstruct
LOAD_SHARE	Load share from RAM to the stack
SHARE	Secret-share TOS
HALT	Store current instruction on stack and halt
COMMIT	Commit to a share on-chain
END_ROUND	Finalize this block
WAIT	Wait for more shares to arrive

Table 5: New network opcodes for CPython bytecode

typing means that local operations on different object types look the same from the VM's perspective, as is the case with the addition gate, which is equivalent to regular Python, except for the return value that is done over the network. Defining the semantics of objects (e.g., what does it mean to add a secret value with a public value) is done on the class-level, which is not shown here. We wrote a separate implementation for secret-sharing primitives, that leverages numpy [11] heavily for optimization of vectors and matrices of shares. The DVM only really needs to know when a communication round is required, as is seen in the example of a multiplication gate.

11.4 Integration with the Bitcoin Blockchain

We now describe in more detail the integration with the Bitcoin blockchain, and what part the federated node has in the consensus. For simplicity, we will describe our system with a single federated node, although a more likely scenario is that several exist.

Consensus Broadcast. In the model, it was mentioned how the reliable broadcast of the blockchain, which is synchronous with eventual delivery and consensus, can be coupled with an unreliable asynchronous broadcast. In order to reduce the

[11] http://www.numpy.org

Addition	Multiplication
LOAD_FAST 0	LOAD_FAST 1
LOAD_FAST 1	LOAD_FAST 3
BINARY_ADD	BINARY_SUBTRACT
LOAD_NAME 0	LOAD_FAST 0
SEND_RESULT_SHARE 1	LOAD_FAST 2
RETURN_VALUE	BINARY_SUBTRACT
	BC_SHARE 2
	HALT
	LOAD_SHARE
	WAIT
	RECONSTRUCT 2
	DUP_TOPX 2
	BINARY_MULTIPLY
	ROT_TWO
	LOAD_FAST 3
	BINARY_MULTIPLY
	BINARY_ADD
	ROT_TWO
	LOAD_FAST 2
	BINARY_MULTIPLY
	BINARY_ADD
	LOAD_FAST 4
	BINARY_ADD
	RETURN_VALUE

Table 6: Secure DVM protocols for addition and multiplication

load on the blockchain, we combine this idea together with that of a federated node, creating a *broadcast with escalation*. Initially, all parties use the (non-resilient) broadcast channel of our internal network. Parties also maintain a direct channel to the federated node, and use that to send a copy of each transaction. The federated node signs each transaction along with a timestamp and stores it in a local database accessible by anyone. To avoid trusting the federated node, each party can check whether its message has been posted and if not, broadcast the transaction to the blockchain before the end of the round, which all parties are also connected to. If later, at some time in the future but before a commit (described below), the federated node deletes a message, the affected party can then broadcast to the blockchain the original time-stamped multi-signed transaction.

Access-control. Recall that access control is defined for each owner o, as a list of services $access[o]$. To implement this in Bitcoin, an owner o creates a pay-to-public-key-hash[12] with her as the input, the addresses of all approved services as the outputs, and one additional output back to the input address. Each output has a minuscule fraction of a bitcoin, so it is negligible. When ever the owner wishes to update the list of permissions, she sends a new transaction that is chained to the output designated to her in the old transaction, using the same logic. This invalidates the previous set of permissions and approves a new set. As an optimization, the owner also sends the federated node the transaction id for local caching. Then, when a service issues a computation request referencing some secret data, both the federated node and the computing parties can validate that it has the appropriate permissions. If a service asks a computation without proper permissions, its payment is forfeited. This protects against potential DoS attacks by the service.

Quorum selection. Using the blockchain as a public randomness source, we can still rely on it to implicitly select the random quorums. The reason we do not trust the federated node with this, unlike some other tasks, is that it would take an expensive cryptographic protocol to have it prove that the selection was indeed random.

Incentives and public identifiability. Recall that at the heart of our protocols lies the incentive scheme. When ever a computing party breaks protocol, it is publicly observable and the party is penalized. This is the core requirement we needed for IC-MPC to hold. There are two types of penalties: either a deposit is burned (i.e., lost forever), or the penalty is used to compensate an honest party. The latter occurs when a computation failed, so charging the service would make our scheme unfair (financially), so instead we force corrupted parties to pay the honest ones.

Contract 2 (shown in Section 4) that validates a computation requires the blockchain to store the transcript and trace the computation using it. Specifically, we need to be able to perform arithmetic operations over finite fields. While Bitcoin has the necessary arithmetic opcodes, in practice several that we require are currently dis-

[12]This is the most common bitcoin transaction type: https://en.bitcoin.it/wiki/Transaction#Pay-to-PubkeyHash

abled (OP_MUL, OP_DIV, OP_MOD). We therefore require the federated node to execute the contract on behalf of the blockchain. The contract's hash is stored in an OP_RETURN transaction on-chain, and the code is public for all parties in the system.

After a computation ends, and the federated node runs the (very public) verification, it settles payments and penalties, by signing for each party if it was honest or not, and refunding the payment to the service if it did not receive an output. Handling the service is simpler (but similar), so its details are omitted. Here we focus on the computing parties. W.l.o.g, we will look at a single computing party Alice (or A) and the federated node F.

A simplified solution starts with Alice broadcasting a *security deposit* transaction (SD), paying to a (2,2) multi-sig address owned jointly by her and F. Alice then prepares three redeeming transactions, and signs them: a time-locked refund transaction (R) paying her back the full deposit after some (long) period of time; a transaction-puzzle (marked SDa) storing $H(r)$ that is redeemable by both F's signature and anyone who can provide the pre-image r; and a Null-data transaction (denoted as SDb) [13] making the funds unspendable. Alice sends transaction (R) to F, the random secret r to *all* other parties, and waits until F signs (R) before sending (SDa, SDb).

After the computation, if Alice was honest, she can create another refund transaction without the time-lock and ask F to sign it, or wait for (R) to become valid. In any case, F should not sign (SDa) or (SDb) unless Alice was malicious. In that case, if the overall computation succeeded (or it failed but all honest parties were compensated), F signs and broadcasts (SDb) to burn Alice's deposit. Otherwise, it signs (SDa) and broadcasts it. It then creates a claim transaction (C) for an honest party (e.g., Bob), signs it and *privately* sends it to him. Notice that this procedure matches exactly the verification and rewarding/penalizing presented in Contract 2.

There are clear inefficiencies in this scheme. For every computation, at least $2n$ new transactions will be added to the blockchain (2 for an honest party, 3 for corrupt).

[13] A Null-data transaction uses OP_RETURN to mark the transaction as invalid.

More importantly, to ensure that the parties do not double-spend their original (SD) transaction, we need to wait until these transactions are confirmed on the blockchain. If we use the best practice of waiting for six confirmations, then it will be an hour before we can even *begin* a computation.

We address these concerns by using an idea similar to payment channels [14]. Instead of having a one-time security deposit as described above, when parties register to the network they lock a large amount, by creating (SD0) and (R0) transactions in the same way as above. The zero at the end marks that this is the state of the security deposit account in the beginning. This is also in line with the register call we described in Contract 1, where parties register by locking a large $deposit_{TOTAL}$ amount.

Figure 4 illustrates this process with an example. Alice deposits 10 BTC in SD0, which she will get back at some point in the distant future (1000 blocks from now). Note that this is the *only* transaction on-chain and that it is not computation-dependent. Now, in each computation, Alice does as before – she generates similar (SD1a, SD1b, R1) transactions (In practice, there should be (R1a) and (R1b), but we omit this detail for simplicity), but with each having two outputs: the actual deposit for this computation (1 BTC) and the remaining 9 BTC. If Alice is found to be cheating, then F either signs (SD1a) and sends a signed (C1) to Bob, or it signs (SD1b). However, F did not broadcast any transaction at this point, and the entire accounting occurred off-chain. In the next computation, Alice will have to build on top of either (SD1a) or (SD1b), which now has 9 BTC left. Otherwise, F should ignore Alice. If Alice was honest, then these transactions are ignored and Alice should use (SD0) as before (i.e., she did not lose her deposit).

The example given in the figure shows two executions in which Alice participated after originally depositing 10 BTC. In the first, Alice cheated and Bob was compensated. In the second, Alice was honest and therefore these transactions are void. When Alice asked F to unregister, she used the output from the first execution to create a transaction (E1) that refunds the remaining 9 BTC. As illustrated in the figure, until Alice either unregisters, cheats sufficiently many times or the original deposit

[14] https://en.bitcoin.it/wiki/Contract

expires, only a single transaction will be logged on the blockchain. Only when the expiry date approaches or if Alice is a serial-cheater, will F and other compensated parties broadcast the chain of transactions (marked in green). Note that if Alice is always honest, then in each sufficiently lengthy interval (e.g., 1000 blocks) she will only ever post two transactions.

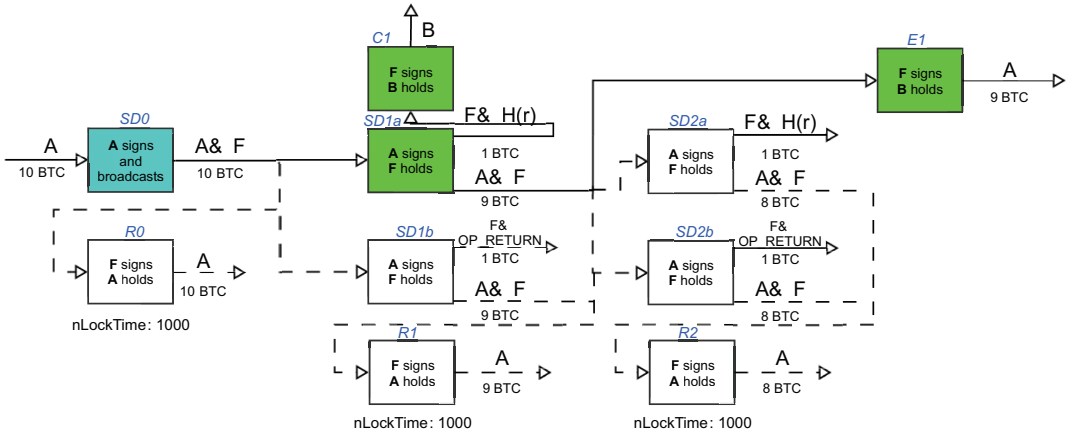


Figure 4: Off-chain and on-chain accounting of incentives.

Long-term storage. In the short term, parties are interacting with the federated node and can visibly see if it breaks protocol. However, in the long term, we would like to use the blockchain to ensure the integrity of the public state (mainly commitments and transcripts), without trusting the federated node. We therefore set a time interval (e.g., 1 day), after which the federated node creates a *merkle tree* (MT) [15] for the current public state. It then creates a Null-data transaction that stores the merkle root and sends it to the blockchain (we call this *committing* the state). In practice, the merkle root is a hash of nested trees, each representing a different portion of the state. For example, we generate a tree for each transcript of a computation that took place that day. We then use these merkle roots as leaves to another merkle tree. We have a similar process for commitments of secrets. With this approach, nodes in the network can always ask the federated node to prove (in logarithmic time) what the state was at any given time in the past.

[15] We assume the reader is familiar with merkle trees.

11.5 Refined Security Analysis

The federated solution imposes some centralization in our otherwise fully decentralized system, but it is likely to be a temporary solution to a short-lived problem [16]. We designed the system in a way that never reveals private data to a federated node and that any deviation is publicly visible. In that sense, parties trust a federated node like miners in a Bitcoin mining pool trust a pool manager. In both cases, there is a hub, which we assume is honest, that collects broadcasted transactions and pays rewards based on (honest) work.

Note that we left open how a federated node is incentivized, but one could assume that this would be based on fees similar to how mining pools operate. In this case, there would be several federated nodes competing for parties' resources, so the operators would have an implicit incentive to remain honest (otherwise parties would switch). Another solution to somewhat reduce the trust is to have several federated nodes simulate F using a standard consensus protocol [50]. Efficiency is not a concern since we are only interested in eventual consensus and this federated sub-network is likely to be small. Similarly, we can remove federation by selecting quorums of verifiers (or validators) and have them reach consensus. This can be done less frequently and with more nodes compared to the size of computing quorums, as it is less time critical.

In any case, there are several solutions to decentralizing this (already limited) amount of federation in the future, either when blockchains are scalable enough, or by rolling out an internal consensus mechanism, which leverages the blockchain for long term consensus.

[16] For the very least, quorums could be selected to do the verification with low overhead.

12 Evaluation

12.1 Empirical Analysis of Quorums

Until recently, achieving large-scale MPC was not practical due to the communication overhead that grows linearly with the number of parties (from a single party's perspective. The overall complexity grows quadratically). Using quorums presented a theoretical solution, but creating them was expensive and re-using them was a problem. In our solution we overcame this expensive pre-processing step and avoided re-using quorums. The only remaining question was deciding on a quorum size.

Ideally we wanted to choose the smallest quorum possible (i.e., 3 parties), but there is a non-negligible probability of selecting 3 corrupted parties for a computation that would break privacy. The case of resiliency is easier since we can identify, penalize, and replace active corruptions. Note that presumably, as the network scales, more (rational) honest parties would join for the monetary rewards it offers, thus reducing the fraction of parties the adversary controls. And yet, to be on the safe side and conform with previous work, we decided to test the commonly used cases of $T = \frac{N}{2}$ and $T = \frac{N}{3}$ *overall* corruptions in the network. We use capital N, T here to distinguish between the complete network and the quorums.

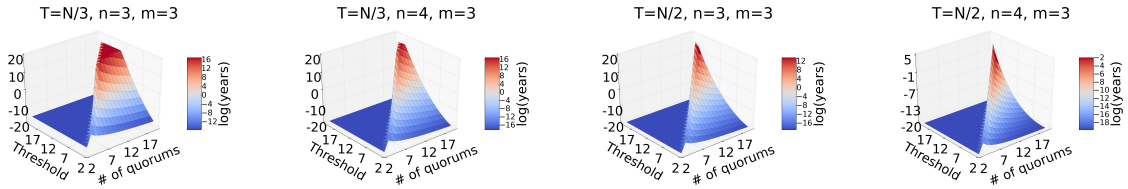
With these parameters fixed, the number of corruptions in a randomly selected quorum follows a binomial distribution, with parameters $n, p \in \{\frac{1}{3}, \frac{1}{2}\}$, where n is the number of parties in the quorum. Assuming we are using a (n, t) secret sharing scheme in the quorum, we can quantify the event of selecting a bad quorum using:

$$Pr[X \geq m] = \sum_{i=m}^{n} \binom{n}{i} p^i (1-p)^{n-i}, \tag{21}$$

where $m := t + 1$.

To provide context to this probability, we assume our system is handling one million requests per-day and check how long it would take (in expectation) to choose a corrupted quorum that leaks the private information of a *single* execution. Just by plugging in numbers, we can see that a relatively large quorum is required (ap-

proximately 100 parties, depending on the secret sharing threshold). To mitigate this, we use a *quorums of quorums* approach. Instead of having a single quorum, we have a small (3 or 4 parties) quorum for each share. Figure 5 illustrates how long the adversary is expected to wait until it can successfully leak information from one execution. Notice that using a $(3, 2)$ secret sharing is much better than $(4, 2)$, namely – it is better to use the maximal $(n, n-1)$ sharing internally, and be more flexible on the external sharing. For example, if we set the internal quorum size to three parties using $(3, 2)$ sharing, it would take approximately three years to leak data, assuming 9 such quorums execute each computation, and the protocol tolerates three corrupted quorums. If we increase the threshold by one (i.e., at most two corrupted quorums are tolerated), it would take 300 years to leak data from a single execution.



(a) (b) (c) (d)

Figure 5: Analysis of time until the first bad quorum is selected.

12.2 Scaling

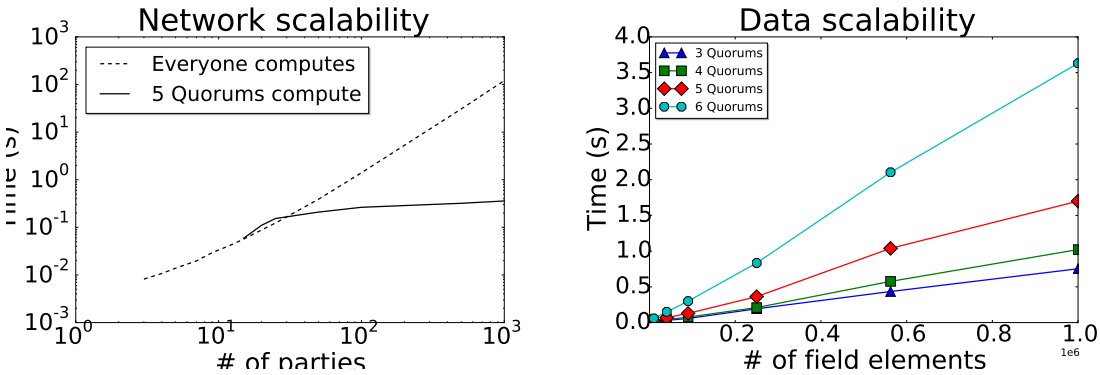
To analyze the performance of our system, we tested it with three to six quorums for each computation, each with three parties using $(3, 2)$ sharing.

Two scenarios were examined: one in which we scale the network, keeping the data fixed at multiplying 10,000 field elements (see Figure 6a); and the other where we scale the number of field elements from one to a million (multiplication). The first graph also shows a simulation of a network-wide MPC computation, i.e., no quorums, as is the case with all previous implementations. Notice the slight overhead in our system due to quorum selection and using two-levels of secret-sharing, which becomes insignificant almost immediately as the network scales beyond 50 nodes.

It is important to note the significance of MPC that scales independently of the

number of nodes. Effectively, this means that in the future we could parallelize computations in the network, as is commonly done in *high performance computing* (or *HPC*). Since our quorums are constant in size, we could let every quorum simulate a processor in a heavy-duty concurrent computation. This implies that having a larger network can actually yield better performance, in contrast to vanilla MPC protocols.

Figure 6b illustrates that adding even a single quorum is costly, especially beyond 5 quorums and with larger data. This is why we ended up choosing 5 quorums as our default configuration.



(a) first (b) second

Figure 6: Network and Data scaling.

12.3 Benchmarks

The results above illustrate how performance is generally affected by changing the number of parties or the size of the data, as was discussed in the first part of this thesis. Another aspect is to test the performance of the improved secure protocols developed in the second part.

For these tests, we have set $n = 3$, which is commonly used to benchmark MPC systems [4], [59]. To evaluate both integers and fixed-point values, multiplication, comparison and division were tested over blocks of 10,000 64-bit secret-shared elements in parallel. The results in Table 7 are the average time in milliseconds over 100 runs.

Mult.	Comparison	Div.	Comparison [29]	Div. [28]
172	211	1813	1448	13029

Table 7: Run-time (in ms) of selected protocols.

In addition to the improved protocols developed earlier, a variant of secure comparison and fixed-point division as proposed in [28] [29] were developed and tested as well. As seen, the new protocols perform significantly better and thus agree with the theoretical complexity analysis.

13 Conclusions

The work presented in this thesis attempts to make a first meaningful step towards scalable secure multi-party computation deployed in practical systems. To this end, cryptographic assumptions that are usually assumed with no justification (e.g., number of corruptions, cost-less consensus broadcast, asynchronous communication) were implemented using means available today, most notably by leveraging a blockchain. This has culminated in the development of an *incentive-compatible* model for MPC, re-visiting the idea of rational parties and proving that incentives can overturn previous negative results on rationality. Instead, it was shown that rationality, when based on incentives, can lead to a highly optimized version of MPC that discourages active corruptions. By design, the system attains other important properties such as financial fairness and output delivery guarantees, as well as public verifiability of results. IC-MPC was also used to develop (nearly) asynchronous MPC that can maintain a high threshold of active corruptions, given that parties are rational. This is essential for any practical deployment of MPC, as a synchronous model could lead to a delay linear in the depth of the circuit, if most of the time is spent waiting for the round to end and the parties' clocks to synchronize. Instead, with asynchronous IC-MPC the delay is at most one round, even for the general case where $t < n$.

Optimizing MPC is the invisible thread connecting the different dots explored in this work. For the first time, the theoretical idea of using quorums was examined in

practice, allowing the number of parties to scale while increasing both security without impairing efficiency. Several chapters were dedicated to creating a new framework for secure computation over the integers and reals (represented as fixed-point numbers). The results show significant improvements compared to the state-of-the-art.

An initial implementation of such a platform, that allows for general-purpose computing and storage of secret information, was developed. This implementation is a significant step forward towards creating a secure cloud system, where owners are in control of their data and they can set permissions on who can query them, without ever observing the raw information directly. The potential of such a system extends to many verticals, such as medical, financial, and even to consumer applications. In an era of Big Data in particular, there is immense potential in providing researchers and organizations with the ability to train statistical models while keeping data private. This will likely allow them to share and consume data like never before.

To conclude, the relation of this work to the emerging field of blockchain research is emphasized. Blockchain technology provides a form of incentive-compatible byzantine agreement (IC-BA). This is the first work to formalize such an economic model for studying the security of secure computation, which is a generalization of BA. Therefore, while the system developed leverages in a black-box manner the blockchain, it also serves as an extension to it, allowing computation with *privacy* and not just *correctness*.

Bibliography

[1] Nakamoto, Satoshi. "Bitcoin: A peer-to-peer electronic cash system." Consulted 1.2012 (2008): 28.

[2] Yao, Andrew C. "Protocols for secure computations." 2013 IEEE 54th Annual Symposium on Foundations of Computer Science. IEEE, 1982.

[3] Ben-David, Assaf, Noam Nisan, and Benny Pinkas. "FairplayMP: a system for secure multi-party computation." Proceedings of the 15th ACM conference on Computer and communications security. ACM, 2008.

[4] Bogdanov, Dan, Sven Laur, and Jan Willemson. "Sharemind: A framework for fast privacy-preserving computations." Computer Security-ESORICS 2008. Springer Berlin Heidelberg, 2008. 192-206.

[5] Team, VIFF Developement. "Viff, the virtual ideal functionality framework." 2009.

[6] Ben-Or, Michael, Shafi Goldwasser, and Avi Wigderson. "Completeness theorems for non-cryptographic fault-tolerant distributed computation." Proceedings of the twentieth annual ACM symposium on Theory of computing. ACM, 1988.

[7] Goldreich, Oded, and Rafail Ostrovsky. "Software protection and simulation on oblivious RAMs." Journal of the ACM (JACM) 43.3 (1996): 431-473.

[8] Goldreich, Oded. "Towards a theory of software protection and simulation by oblivious RAMs." Proceedings of the nineteenth annual ACM symposium on Theory of computing. ACM, 1987.

[9] Goldreich, Oded, Silvio Micali, and Avi Wigderson. "How to play any mental game." Proceedings of the nineteenth annual ACM symposium on Theory of computing. ACM, 1987.

[10] Chaum, David, Claude CrÃľpeau, and Ivan Damgard. "Multiparty unconditionally secure protocols." Proceedings of the twentieth annual ACM symposium on Theory of computing. ACM, 1988.

[11] Rabin, Tal, and Michael Ben-Or. "Verifiable secret sharing and multiparty protocols with honest majority." Proceedings of the twenty-first annual ACM symposium on Theory of computing. ACM, 1989.

[12] DamgÃěrd, Ivan, et al. "Practical covertly secure MPC for dishonest majorityâĂŞor: Breaking the SPDZ limits." Computer SecurityâĂŞESORICS 2013. Springer Berlin Heidelberg, 2013. 1-18.

[13] Beaver, Donald. "Efficient multiparty protocols using circuit randomization." Advances in CryptologyâĂŤCRYPTOâĂŹ91. Springer Berlin Heidelberg, 1992.

[14] Dani, Varsha, et al. "Brief announcement: breaking the o (nm) bit barrier, secure multiparty computation with a static adversary." Proceedings of the 2012 ACM symposium on Principles of distributed computing. ACM, 2012.

[15] Dani, Varsha, et al. "Quorums quicken queries: Efficient asynchronous secure multiparty computation." Distributed Computing and Networking. Springer Berlin Heidelberg, 2014. 242-256.

[16] Boyle, Elette, Shafi Goldwasser, and Stefano Tessaro. "Communication locality in secure multi-party computation." Theory of Cryptography. Springer Berlin Heidelberg, 2013. 356-376.

[17] WOOD, DR GAVIN. "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER." (2014).

[18] Andrychowicz, Marcin, et al. "Secure multiparty computations on bitcoin." Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014.

[19] Bentov, Iddo, and Ranjit Kumaresan. "How to use bitcoin to design fair protocols." Advances in CryptologyâĂŞCRYPTO 2014. Springer Berlin Heidelberg, 2014. 421-439.

[20] Halpern, Joseph, and Vanessa Teague. "Rational secret sharing and multiparty computation." Proceedings of the thirty-sixth annual ACM symposium on Theory of computing. ACM, 2004.

[21] Abraham, Ittai, et al. "Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation." Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing. ACM, 2006.

[22] Schultz, David A., Barbara Liskov, and Moses Liskov. "Mobile proactive secret sharing." Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing. ACM, 2008.

[23] Ishai, Yuval, Rafail Ostrovsky, and Vassilis Zikas. "Secure Multi-Party Computation with Identifiable Abort." Advances in CryptologyâĂŞCRYPTO 2014. Springer Berlin Heidelberg, 2014. 369-386.

[24] Zyskind, Guy, Oz Nathan, and Alex'Sandy Pentland. "Decentralizing Privacy: Using Blockchain to Protect Personal Data." Security and Privacy Workshops (SPW), 2015 IEEE. IEEE, 2015.

[25] DamgÃěrd, Ivan, et al. "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation." Theory of Cryptography. Springer Berlin Heidelberg, 2006. 285-304.

[26] Nishide, Takashi, and Kazuo Ohta. "Multiparty computation for interval, equality, and comparison without bit-decomposition protocol." Public Key CryptographyâĂŞPKC 2007. Springer Berlin Heidelberg, 2007. 343-360.

[27] Lipmaa, Helger, and Tomas Toft. "Secure equality and greater-than tests with sublinear online complexity." Automata, Languages, and Programming. Springer Berlin Heidelberg, 2013. 645-656.

[28] Catrina, Octavian, and Amitabh Saxena. "Secure computation with fixed-point numbers." Financial Cryptography and Data Security. Springer Berlin Heidelberg, 2010. 35-50.

[29] Catrina, Octavian, and Sebastiaan De Hoogh. "Improved primitives for secure multiparty integer computation." Security and Cryptography for Networks. Springer Berlin Heidelberg, 2010. 182-199.

[30] Reistad, Tord Ingolf. A General Framework for Multiparty Computations. Diss. Norwegian University of Science and Technology, 2012.

[31] Toft, Tomas. "Constant-rounds, almost-linear bit-decomposition of secret shared values." Topics in CryptologyâĂŞCT-RSA 2009. Springer Berlin Heidelberg, 2009. 357-371.

[32] Pullonen, Pille, and Sander Siim. "Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations." Financial Cryptography and Data Security. Springer Berlin Heidelberg, 2015. 172-183.

[33] Aliasgari, Mehrdad, et al. "Secure Computation on Floating Point Numbers." NDSS. 2013.

[34] Kamm, Liina, and Jan Willemson. "Secure floating point arithmetic and private satellite collision analysis." International Journal of Information Security 14.6 (2015): 531-548.

[35] Krips, Toomas, and Jan Willemson. "Hybrid model of fixed and floating point numbers in secure multiparty computations." Information Security. Springer International Publishing, 2014. 179-197.

[36] Liedel, Manuel. "Secure distributed computation of the square root and applications." Information Security Practice and Experience. Springer Berlin Heidelberg, 2012. 277-288.

[37] Damgard, Ivan, and Rune Thorbek. "Efficient Conversion of Secret-shared Values Between Different Fields." IACR Cryptology ePrint Archive 2008 (2008): 221.

[38] Cramer, Ronald, Ivan DamgÃěrd, and Yuval Ishai. "Share conversion, pseudorandom secret-sharing and applications to secure computation." Theory of Cryptography. Springer Berlin Heidelberg, 2005. 342-362.

[39] Bentov, Iddo, and Ranjit Kumaresan. "How to use bitcoin to design fair protocols." Advances in CryptologyâĂŞCRYPTO 2014. Springer Berlin Heidelberg, 2014. 421-439.

[40] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In Financial Cryptography, 2014

[41] I. Eyal. The MinerâĂŹs Dilemma. In IEEE Symposium on Security and Privacy, 2015.

[42] Bonneau, Joseph, et al. "SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies." (2015).

[43] Kosba, Ahmed, et al. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. Cryptology ePrint Archive, Report 2015/675, 2015. http://eprint. iacr. org, 2015.

[44] Songhori, Ebrahim M., et al. "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits." IEEE S & P. 2015.

[45] Liu, Chang, et al. "Oblivm: A programming framework for secure computation." (2015).

[46] Rastogi, Ayush, Matthew Hammer, and Michael Hicks. "Wysteria: A programming language for generic, mixed-mode multiparty computations." Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014.

[47] Liu, Chang, et al. "Automating efficient RAM-model secure computation." Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014.

[48] Shamir, Adi. "How to share a secret." Communications of the ACM 22.11 (1979): 612-613.

[49] Forges, FranÃğoise. "An approach to communication equilibria." Econometrica: Journal of the Econometric Society (1986): 1375-1385.

[50] King, Valerie, et al. "Scalable leader election." Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm. Society for Industrial and Applied Mathematics, 2006.

[51] Dwork, Cynthia. "Differential privacy." Encyclopedia of Cryptography and Security. Springer US, 2011. 338-340.

[52] Bitansky, Nir, et al. "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again." Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. ACM, 2012.

[53] Damgård, Ivan, Jesper Buus Nielsen, and Antigoni Polychroniadou. "On the Communication required for Unconditionally Secure Multiplication."

[54] Jakobsen, Thomas. Secure multi-party computation on integers. Diss. Aarhus Universitet, Datalogisk Institut, 2006.

[55] Veugen, Thijs, et al. "Secure comparison protocols in the semi-honest model." Selected Topics in Signal Processing, IEEE Journal of 9.7 (2015): 1217-1228.

[56] Kendler, Ethan Heilman Alison, Aviv Zohar, and Sharon Goldberg. "Eclipse Attacks on BitcoinâĂŹs Peer-to-Peer Network."

[57] Blakley, George Robert. "Safeguarding cryptographic keys." afips. IEEE, 1899.

[58] Damgård, Ivan, and Rune Thorbek. "Non-interactive proofs for integer multiplication." Advances in Cryptology-EUROCRYPT 2007. Springer Berlin Heidelberg, 2007. 412-429.

[59] Zhang, Yihua, Aaron Steele, and Marina Blanton. "PICCO: a general-purpose compiler for private distributed computation." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013.

[60] Canetti, Ran. "Universally composable security: A new paradigm for cryptographic protocols." Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on. IEEE, 2001.

[61] Aumann, Yonatan, and Yehuda Lindell. "Security against covert adversaries: Efficient protocols for realistic adversaries." Theory of cryptography. Springer Berlin Heidelberg, 2007. 137-156. APA

[62] Roy, Partha Sarathi, et al. "An Efficient t-Cheater Identifiable Secret Sharing Scheme with Optimal Cheater Resiliency." IACR Cryptology ePrint Archive 2014 (2014): 628.

[63] Dvir, Zeev, and Amir Shpilka. "Noisy interpolating sets for low degree polynomials." Computational Complexity, 2008. CCC'08. 23rd Annual IEEE Conference on. IEEE, 2008.

[64] Ben-Or, Michael, Ran Canetti, and Oded Goldreich. "Asynchronous secure computation." Proceedings of the twenty-fifth annual ACM symposium on Theory of computing. ACM, 1993.

[65] Shen, Emily, et al. "Cryptographically Secure Computation." Computer 48.4 (2015): 78-81.

[66] Yu, Ching-Hua. "Sign Modules in Secure Arithmetic Circuits." IACR Cryptology ePrint Archive 2011 (2011): 539.

[67] Garay, Juan, Aggelos Kiayias, and Nikos Leonardos. "The bitcoin backbone protocol: Analysis and applications." Advances in Cryptology-EUROCRYPT 2015. Springer Berlin Heidelberg, 2015. 281-310.

[68] Eyal, Ittay, et al. "Bitcoin-ng: A scalable blockchain protocol." arXiv preprint arXiv:1510.02037 (2015).

[69] Luu, Loi, et al. "SCP: A Computationally-Scalable Byzantine Consensus Protocol For Blockchains."

[70] Bonneau, Joseph, Jeremy Clark, and Steven Goldfeder. "On Bitcoin as a public randomness source." URL https://eprint. iacr. org/2015/1015. pdf (2015).

[71] Zyskind, Guy, Oz Nathan, and Alex Pentland. "Enigma: Decentralized Computation Platform with Guaranteed Privacy." arXiv preprint arXiv:1506.03471 (2015).

[72] Ercegovac, M. D. "T. Lang Digital Arithmetic." (2003).

[73] Bar-Ilan, Judit, and Donald Beaver. "Non-cryptographic fault-tolerant computing in constant number of rounds of interaction." Proceedings of the eighth annual ACM Symposium on Principles of distributed computing. ACM, 1989.