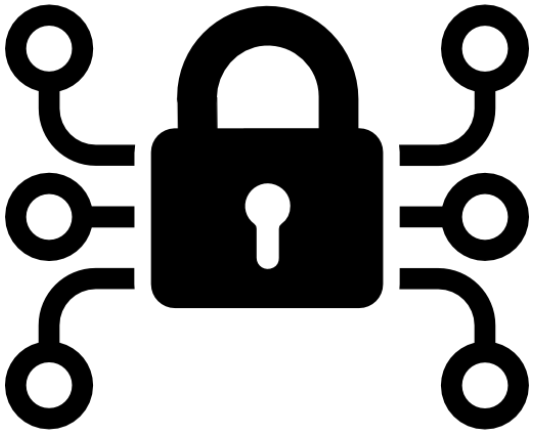
Secure Multi-Party Computation for General Adversary Structures

Danish title: Secure Multi-Party Computation for General Adversary Structures

Authors
Mikkel Skafsgaard Berg - 201907696
Sebastian Dige Bertelsen - 201906062
Lucas Emil Borlund Orellana - 201905747

Advisors
Ivan Damgård
Luisa Siniscalchi



Bachelor report (15 ECTS) in Computer Science
Department of Computer Science
Aarhus University
June 8, 2022

Contents

Abstract

Secure multi-party computation (MPC) is a subfield within cryptography dedicated to solving a wide array of problems that involves a desire for secrecy while avoiding a single point of trust. The purpose of this paper is to provide the theory behind and implementation of a selection of protocols for MPC that are secure against both passive and active corruption. The central adversary is described using general adversary structures which gives relatively simple proofs for security compared to typical threshold models. This simplicity is illustrated in the paper and the main drawback of the protocols, their poor scalability, is addressed as well.

1 Introduction

In this paper, we present and implement a simple and intuitive protocol for multi-party computation which guarantees perfect information-theoretical security against misbehaving participants in general settings. Using the protocol, any function represented as a series of additions and multiplications can be computed without having to rely on a trusted party. The simplicity of the protocol means that proving security is relatively uncomplicated and that the protocol runs well for small numbers of parties, but this comes at the cost of poor scalability. This issue is addressed in the later sections of the paper. The work we present is inspired mainly by Maurer [1], using a framework presented in his paper to generalize the corruption thresholds typically used in MPC protocols, and using his protocols as the basis for the ones we present.

Section 2 and 3 of the paper details the theory and implementation of a protocol for MPC in the passive adversary setting. Section 4 and 5 in turn describes a protocol for MPC in the active adversary setting. Section 6 presents a number of performance tests for the implementation. Section 7 details considerations for future work on the presented protocols.

1.1 Multiparty Computation

Multiparty computation solves the problem of letting a number of parties securely compute a function of their inputs or *secrets*, such that no party learns the secret of any other party, even when some of the parties do not follow protocol. In this paper, the function to be computed will be represented as a series of additions and multiplications, which is sufficient to represent any function.

In general, the series of additions and multiplications will be computed using three sub protocols as follows: Whenever a party's secret is needed it is shared using a secret sharing protocol. Any linear function such as addition can then be done locally using the secret shared values [1]. Multiplication in turn is executed using a multiplication protocol. Once all the necessary additions and multiplications have been executed the result needs to be reconstructed using a reconstruction protocol. The result of the reconstruction protocol is that each party has the function of the secrets.

1.2 Adversaries

In the protocols we will use the notion of an adversary to model misbehaving parties. The adversary has the capability to corrupt a subset of the parties, where we will distinguish between active and passive corruption to specify the ways a party can misbehave. A party that is passively corrupted does not divert from the protocol, but all information known by the party is also known by the adversary. A party that is actively corrupted can behave arbitrarily, specifically, it can divert from the protocol in any possible manner.

1.3 Security

Protocols typically seek to guarantee either computational or information-theoretical security. Here, computational security means security against an adversary with some limit on his computational power, while information-theoretical security means security against an adversary with unbounded computational power. Often, the security of a protocol can be broken with some negligible probability. In this paper we will instead seek to guarantee perfect information-theoretical security, which means that there is a probability of zero

for the security of the protocol to be broken [1].

Security in MPC means that the secret sharing scheme and the multiplication protocol we present will need to satisfy two conditions, correctness and secrecy [2]. These are defined as follows:

Secrecy: The adversary does not learn any information that he did not know beforehand about the secrets of the parties.

Correctness: Correctness means that the honest parties stay in a correct state. The precise definition of correctness depends on the protocol at hand, and will be presented along with the respective protocols.

2 Theory: Passive participants

In this section we present three protocols for the passive adversary setting: A protocol for secret sharing, a protocol for multiplication and a protocol for reconstruction.

2.1 Passive adversaries

In the passive setting, the adversary must follow the given protocol and cannot alter the inputs of the corrupted parties after the protocol has started. Thus, the worst a passive adversary can do is try to reconstruct the secret of other parties, using the intermediary results obtained while following the given protocol. Therefore, the secrecy condition of the protocols depend on whether or not the adversary can learn secrets, which the adversary was not supposed to know, from the intermediary results [1].

To be more precise about which set of parties are allowed to know a secret and which set of parties are not, we need some definitions. We call a subset of the parties that can reconstruct a given secret a *qualified* set for that secret. On the other hand, an *ignorant* set is a subset of the parties, which is unable to reconstruct the secret. We define the *access structure*, denoted Γ, as the set of all qualified party subsets, and the *secrecy structure*, denoted Σ, as the set of all ignorant subsets. From now on we specify the access structure by its minimal elements, since any superset of a qualified set is also qualified, and the secrecy structure by its maximal elements, since any subset of an ignorant set is also ignorant.

The adversary in the passive case is specified by the secrecy structure since the capabilities of the adversary is limited to corrupting exactly one of the sets in the secrecy structure. In the passive case this means that the adversary must try to obtain information outside of the information initially known by the parties in the chosen set, while following the protocol.

The secrecy structure plays a significant role in both the secret sharing and multiplication protocol since we want to ensure that no set in the secrecy structure can reconstruct the secret, as that would allow the adversary to pick that set and break secrecy. For passive adversaries the secrecy structure is specified by the following condition:

$$P \notin \Sigma \sqcup \Sigma \tag{1}$$

Here, P is the set of all parties participating in the protocol and the relation $\sqcup$ on two structures Π_1 and Π_2 is defined as the structure containing all unions of one element from Π_1 and one from Π_2:

$$\Pi_1 \sqcup \Pi_2 := \{S_1 \cup S_2 : S_1 \in \Pi_1, S_2 \in \Pi_2\}$$

Condition (1) can be understood informally as the condition that no two sets in the secrecy structure must contain all parties and is sufficient and necessary for passive adversaries, as proven by Hirt and Maurer [3].

Condition (1) is general in that it allows sets of different size in the secrecy structure. This is different from other results concerning MPC's that are defined using corruption thresholds. In the passive case these typically say that at most t parties can be corrupted where t is the largest integer less than $\frac{n}{2}$ [4][5]. Corruption thresholds can easily be imposed on the secrecy structure, but structures can be used to model

even more adversaries. For example, in a 5-party setting, if it were known that parties 4 and 5 could not collude with any other party, then the secrecy structure $\Sigma = \{\{1,2,3\},\{4\},\{5\}\}$ could be used to describe the adversary. This would not be allowed using the t threshold, but is allowed by condition (1).

2.2 Secret-sharing scheme

Using the secrecy and access structures we can define a secret-sharing scheme. A secret-sharing scheme consists of two elements: A secret sharing protocol and a reconstruction protocol. The secret sharing scheme allows a party to share its secret among the parties, such that no ignorant set $T_i \in \Sigma$ can reconstruct it, while all qualified sets $S_i \in \Gamma$ have the ability to reconstruct it. In this section we will first present the secret-sharing and reconstruction protocols, then argue for the correctness and secrecy of the secret sharing scheme.

2.2.1 Secret sharing

The secret-sharing protocol is used to share one secret with all of the parties such that the sum of the shares make up the secret. The first step of secret-sharing is splitting the individual party's secret, s, into k shares, $s_1, \ldots, s_k$, where k is the size of the secrecy structure. This is done by first selecting the first *k-1* shares $s_1, \ldots, s_{k-1}$ randomly from a domain $\mathcal{D}$. We then define s_k as

$$s_k = s - \Sigma_{i=1}^{k-1} s_i$$

That is, we set the last share of the secret, s_k, to equal the difference between the secret and the sum of all the other shares. That way, the sum of all the shares is equal to the secret. Note that the last share also needs to stay within the domain, which is ensured in different ways depending on the chosen domain. For example if the domain is the integers $\mathcal{Z}_{\mathcal{D}} = 1, \ldots, |\mathcal{D}|$ the modulo operation can be used with the divisor being $|\mathcal{D}|$ to ensure that the last share stays within the bounds of the domain.

After creating the shares, we now share these with the different parties. To maintain secrecy while doing so, we need to ensure that for each set in the secrecy structure $\Sigma = \{T_1, \ldots, T_k\}$, at least one share is not sent to any party in that set. To do this, we define $\overline{T}_i = P \setminus T_i$, that is, the complement of the i'th element of the secrecy-structure, in the set of all parties. We refer to the set of all $\overline{T_i}$ as the *distribution set*. We can now send share s_i to each party in the set $\overline{T}_i$. In summary we have:

Secret-Sharing Protocol $(\Sigma, \mathcal{D})$

1. Split secret s into k shares s_i. Select *k-1* shares $s_1, \ldots, s_{k-1}$ randomly from the domain $\mathcal{D}$. Let $s_k = s - \Sigma_{i=1}^{k-1} s_i$
2. Send share s_i to all parties in the set $\overline{T}_i$

Note, that since secret sharing is used extensively in MPC and depends on k, the way we have defined k does not allow for a scalable solution, since k could increase very quickly as a function of n. For example, if the sets in the secrecy structure are all the largest subsets of size $t = \lceil \frac{n}{2} \rceil - 1$, then $k = \binom{n}{t}$ which is exponential in n, since t is a fixed fraction of n [1]. What we gain at the cost of scalability is that we can argue thoroughly for the correctness and secrecy of the protocols using this way of secret sharing.

2.2.2 Reconstruction

After a secret has been shared, each party has some of the shares needed to compute the actual secret. For a qualified set of parties to reconstruct a secret they each need to end up with all of the shares of the secret. To achieve this, the parties in the qualified set can simply send all of their shares to each other party in the qualified set. After this, the parties can sum up the shares to obtain the secret.

Note that the reconstruction protocol is used at the end of the entire MPC protocol to reconstruct the

function of the input secrets. Here, the entire set of parties is the qualified set running the reconstruction protocol.

2.2.3 Proofs

Correctness
Correctness of the secret sharing scheme means that any qualified set $S \in \Gamma$ is able to reconstruct a shared secret $s = s_1, \ldots, s_k$. To argue for this, note that for any set $S \in \Gamma$, for each set $T_i \in \Sigma$, S contains a party not in T_i. If this were not the case, S would be a subset of T_i meaning that it would not be qualified. Since S contains parties outside of all $T_i \in \Sigma$, it receives every share s_i and is able to reconstruct the secret.

Secrecy
Assuming that condition (1) is fulfilled, what we want to prove is that no set in the secrecy structure Σ can reconstruct a shared secret. We can see that this is true, since for any set $T_i \in \Sigma$, at least one share would be missing, since share i is sent to the complement set $\overline{T}_i$. This means that no set $T_i \in \Sigma$ receives all the shares of the secret. Furthermore, since the shares are chosen at random there is no way to guess the secret unless all shares are present. Consider, for example, the case, where we have a secret s, which we divide into three shares, $s = s_1 + s_2 + s_3$. If we then send s_1 and s_3 to another party, they have no way to deduce s_2, as the shares are chosen randomly from a domain. Most importantly, there is no way to verify your guess. All numbers in the domain are just as likely to be correct. In other words, the protocol is perfectly information-theoretically secure, since brute force attacks are impossible, regardless of the amount of adversarial computing power.

It may seem excessive to use the notion of a secrecy structure to guarantee security. To see the practicality of this, consider the following insecure way of secret sharing: For this approach, the indices of the shares that a party p_i receives is the closed interval $[i, (i+l-1) \bmod \mathrm{k}]$, where l is the number of shares sent to each party. To see why this approach breaks secrecy, let us look at a concrete counter-example. In this example the number of parties, n, is 5, l is 6 and the secrecy structure is all possible subsets of the parties of size 2, resulting in $k = 10$. The secret to be shared is $s = s_1, \ldots, s_k$. After the secret has been shared using the incorrect approach, we see that p_1 and p_5 hold the shares:

$$p_1 : s_1, s_2, s_3, s_4, s_5, s_6$$

$$p_5 : s_5, s_6, s_7, s_8, s_9, s_{10}$$

Now if p_1 and p_5 were to collude, they would have all the shares of the secret, despite the set p_1, p_5 being in the secrecy structure. As such, this naive approach violates the secrecy condition.

2.3 Multiplication

As a precondition to the multiplication protocol, two values s and t must be shared among the parties. The postcondition, in turn, is that each party has shares of the product st. Notice that this postcondition does not mean that everyone receives the product st, but rather that if all parties were to send out their shares of st, only then would they be able to compute the product. This allows executions of the multiplication protocol to be easily chained together before the final result is computed using the reconstruction protocol.

The product st can be split into the following sum:

$$st = \left(\sum_{i=1}^{k} s_i\right) \cdot \left(\sum_{j=1}^{k} t_j\right) = \sum_{i=1}^{k}\sum_{j=1}^{k} s_i t_j$$

which yields a sum of k^2 terms. For each term $s_i t_j$ in the above sum there exists *at least* one party who knows both s_i and t_j, given our secret-sharing protocol [1]. As such, we need a deterministic algorithm to determine which parties should compute these duplicated terms, such that no copies are added to the resulting product. We achieve this, in part, by partitioning the set $\{(i,j) : 1 \leq i,j \leq k\}$ into n sets $U_1, ..., U_n$ such that for all $(i,j) \in U_m$ we have $m \in \overline{T_i} \cap \overline{T_j}$.

We refer to this partition as U. Using the partition U, each party p_m can compute the terms assigned to him, and sum these together to achieve a value v_m. The party p_m is qualified to both of the factors of these terms exactly because $m \in \overline{T_i} \cap \overline{T_j}$. Using the secret sharing protocol, these values can now be shared among the parties. The parties can then add together the shares received, and save them as the shares of st. To summarize:

Secure Multiplication Protocol (Σ)

Precondition: Two values $s = \sum_{i=1}^{k} s_i$ and $t = \sum_{i=1}^{k} t_j$ are shared among the players

Postcondition: st is shared among the players

1. Partition the set $\{(i,j) : 1 \leq i,j \leq k\}$ into n sets $U_1, ..., U_n$ such that for all $(i,j) \in U_m$ it holds that $m \in \overline{T_i} \cap \overline{T_j}$
2. Each party p_m (for $1 \leq m \leq n$) computes $v_m := \sum_{(i,j)\in U_m} s_i t_j$ and secret shares v_m among all parties.
3. Each party locally adds shares received in step 2.

Correctness
Correctness of the multiplication protocol means that the postcondition is satisfied. To argue that this is the case, it is evident from step 1, that if the partition is created correctly, then the sum $\sum_{m=1}^{n} v_m$ will be equal to st. Furthermore, for any $(i,j) \in U_m$ the party m is guaranteed to have the shares s_i and t_j exactly because $m \in \overline{T_i} \cap \overline{T_j}$ and because of the way the secret sharing protocol works. Thus, the correctness of the partitioning and subsequent sharing of st follows from the correctness of the secret sharing scheme.

Secrecy
To argue for secrecy, note that the multiplication protocol uses secret sharing, for which we already presented arguments for secrecy. The only other step in the protocol, in which information is shared, is step 2. In this step the parties exchange some information about their shares and this creates potential issues that need to be kept in mind. For example, if a party holds the share s_i and does not hold the share t_j but knows that another party will only share the term $s_i t_j$ (this is possible since some parties might only have one term in their subset of the partition, and all parties know the partition), then the value t_j could be reconstructed if the terms were simply sent out. We handle this issue by using our secret sharing protocol for the sharing of the v values, resulting in this protocol being perfectly information-theoretically secure.

3 Implementation: Passive participants

3.1 Implementation Language: Go

All code for this paper was written in Go, as this language handles multi-threading and concurrency especially well. A link to the repository containing the code can be found in appendix A.1. We will not be going into

much detail about the unusual parts of the Go syntax, apart from the following:

- *Struct.* A Go struct is similar in every relevant way to that of a C struct and resembles an object in object oriented programming.
- *Slice.* A slice is a pointer to (or view of) an underlying array.
- *Channel.* A channel is assigned a type. Variables of that type can be sent into the channel and pulled out at a later time by anyone with access to said channel.
- *Goroutine.* A goroutine spawns a new thread, branching on some function, after which the control flow of the new thread and the source thread proceed independently.

3.2 Previous implementations

Before reaching the final implementation, we dabbled with other less abstract implementations. The primary differences are that secret sharing was done for each sub-expression, rather than preemptively secret sharing all input secrets between all involved parties, and reconstructing the function of the secrets at the end of the protocol. Altogether this yielded a non-modular design and made it harder to chain together multiple sub-expressions. In the following, we sought to avoid this.

3.3 Structure and Approach

The primary parts of abstraction consists of Parties and a Network, which represent participants of the protocol, some of which may be corrupted, and a simulated network, respectively. In practice, we will be using n parties and one network. The Network enables communication between parties and functions in all important aspects as a proper network, but uses channels instead of actual network connections for simplicity. A party has a number of fields including a party number, a secret, slices for storing shares and slices for storing intermediary results. When the network is started, the secrecy structure and the adversary structure (which we describe later) is provided in order to check that they adhere to the conditions required for correctness.

The way the addition and multiplication protocol is used in our implementation simulates the usage of logic gates. As such, a conceptual circuit can be built out of a range of addition- and multiplication circuits, feeding on each others output, to eventually produce the result of the input function. We model these circuits using a slice of Instructions.

Each Instruction is specified by its type, the indices of its operands and its instruction number. The type of an instruction is either an addition or a multiplication. The operands to an instruction correspond to a node in the modelled circuit, with the first n nodes representing the input secrets of the parties. Each party has slices (called intermediary results) for storing the results of each Instruction, and the instruction number is used to store the result of an Instruction in the correct intermediary result. As an example the circuit evaluating the expression $ab + cd$ is modelled by three instructions. One for computing ab one for computing cd and one for computing the sum of the results of the first two instructions.

We opted for this approach over the previous implementations as it is better suited for handling longer expressions and is more concise in the sense that we only need to call the main function once, rather than iteratively depending on the input expression. Thus our main function encapsulates the sub-protocols of addition and multiplication in the desired modular fashion. Additionally, it makes our implementation act more like a secure calculator, which potential MPC applications can be built on top of.

The main function of the code is *MPCProtocol()* [1], which takes a slice of instructions and a slice of parties as input. The slice of instructions must be created beforehand for the function one wants to compute with MPC. To fulfill the precondition of multiplication, and make addition of secrets possible, *MPCProtocol()* secret shares all parties' secrets preemptively, such that all parties hold shares of the secrets of all parties. It

[1] We use '()' to distinguish functions from variables and structs

then iterates over the instruction slice and starts a goroutine for each party based on the instruction type. Finally, it reconstructs the result of the computation at each party and terminates the protocol.

Our implementation uses goroutines extensively such that all parties can proceed in a concurrent fashion. To ensure that no party gets too far ahead in the protocol we use a library implementation of waitgroups. Goroutines can be added to a waitgroup, after which the caller function can wait for all the spawned routines to terminate. Thus, whenever our protocol requires that all parties receive a series of messages from the other parties, the main protocol spawns a sender function and a receiver function for each party, adds all the receiver functions to a waitgroup, and waits for them to terminate.

Our development process followed the Test Driven Development (TDD) principles (for more information, see Baerbak [6]), to ensure a fast and sturdy production. As such, a series of tests serves as the backbone of our implementation, ensuring that the theoretical protocols were implemented correctly, and acting as a safety net for any refactoring we did during the project. We chose not to model the adversary for the passive case, as a passive adversary can only try to break secrecy, and we rely on our proofs for secrecy to protect against this.

3.4 Secret sharing

For our secret sharing scheme, some domain is chosen, depending on the intended application, and distributed to all parties such that they all agree on this parameter. This protocol is information-theoretically secure as explained in the theory section. As such, from a security perspective, any domain with two elements or more is sufficient to prevent brute force attacks. In the implementation we decided to work with integer domains as specified by a number d, such that the domain is $\mathcal{D} = 1, \ldots, d$, and the modulo operation can be used to stay within the domain. After choosing the domain, each party chooses a secret at random from the domain. The secret is chosen randomly to simulate the arbitrary choice of secret input that a party may have. Each party has a copy of the distribution set determined by the secrecy structure, so they can easily determine how shares should be distributed.

Apart from these design choices our secret sharing scheme follows the approach discussed in the corresponding theory section. We have chosen to represent shares as slices of size k since we almost always want to send more than one share at a time. The indices of the slices correspond exactly to the sets in the secrecy structure (and therefore also the distribution set), such that the share corresponding to T_i in the secrecy structure is located at index i in the share slice. This way of representing shares also allows to easily add together slices as all that is needed to add together two shared values is adding the two corresponding slices entry-wise.

3.5 Multiplication

For the implementation of multiplication, we use the aforementioned partition, U, in tandem with a greedy algorithm, as our choice of a deterministic algorithm, that assigns the computation of (i, j) to the first party encountered in the computed set $\overline{T_i} \cap \overline{T_j}$. The different subsets of the partition $U_1, \ldots, U_n$ are created in *createPartition()* and each party receives his subset as part of *startNetwork()*.

An obvious complication with this approach is that lower indexed parties will get responsibility for more shares than higher indexed ones, leading to an uneven work distribution. However, since the parties only need to send a single sum as a result of the work distribution and computing a large sum locally takes a negligible amount of time, this has no real consequences for the total running time of the protocol.

The implemented multiplication protocol closely follows the outlined protocol of the theory section. First, each party computes the sum of the terms in their assigned subset and secret shares this sum between all parties, as specified in step 1 of the protocol. When a party receives a share it is added entry-wise to a temporary slice, and when all of the shares have been received this slice is copied into the intermediary result.

3.6 Reconstruction

After all the Instructions have been executed, each party has shares of the result stored in the last slice of their intermediary results. As the first step of reconstruction, these shares are send out. When received, they are sorted for uniqueness such that every party ends up with a single copy of each share. When all unique shares have been collected, they are summed up. Lastly, the modulo operation is applied to achieve the result of the entire computation.

4 Theory: Active participants

4.1 Active adversaries

Until now we have assumed that the parties participating in the protocol are well behaved and executes the protocol as told without misbehaving in any way. We now consider the case, where some subset of the parties have the ability to actively go against the protocol. To protect against such active corruption, we need to make changes to both the secret sharing scheme and the multiplication protocol. The new secret sharing scheme will be called Verifiable Secret Sharing (VSS) while the multiplication protocol will be called Robust Multiplication.

We now elaborate on the adversary's corruption capabilities in the active setting. As mentioned earlier, the adversary's capabilities can be described by a threshold determining the allowed number of corrupted parties, but can also be described more generally using structures. The structure for passive corruption was the secrecy structure, and we now introduce the *adversary structure* Δ for active corruption. The adversary structure is the set of subsets of the parties that can be actively corrupted and is a subset of the secrecy structure. The adversary has the capability to choose one of the subsets in this structure to corrupt, while maintaining the ability to passively corrupt one of the sets in the secrecy structure. This model is known as a mixed adversary model [1]. The goal of the adversary is either to break the correctness of the protocol for any honest party, using the actively corrupted parties, or to obtain information that the passively corrupted parties are not qualified to know, thus breaking secrecy.

The active protocol requires stricter conditions than (1) to guarantee secrecy and correctness. Specifically, we require the following two conditions on the secrecy structure Σ and the adversary structure Δ to be able to guarantee secrecy and correctness:

$$P \notin \Sigma \sqcup \Delta \sqcup \Delta \tag{2}$$

$$P \notin \Sigma \sqcup \Sigma \sqcup \Delta \tag{3}$$

The last condition is the strongest of the three conditions we have required, and is both sufficient and necessary for general perfect information-theoretically secure MPC [1].

To ensure correctness in the following protocols, we also need some system to ensure agreement among the honest parties when some value is sent out, and that the value received is correct when the sender is honest. This is achieved by relying on a broadcast channel, or a simulation thereof.

4.2 Verifiable Secret Sharing (VSS)

Similar to the secret-sharing scheme in the passive adversary setting, VSS consists of two protocols: VSS Share and VSS Reconstruct. These accomplish the same task as in the passive cases but are secure against corrupted parties deviating from the protocols. We will again first present the two protocols then argue for the correctness and secrecy of the VSS scheme.

4.2.1 VSS Share

One of the most obvious types of attacks we will guard against is a dealer sending different values for the same share to different parties during the secret-sharing phase. To protect against this type of attack, we

add a verification phase, for making sure that all parties, who receive the same share, also receive the same values for said share. For this protocol, we start off by splitting the secret into k shares and sending out those shares, just as in the protocol for passive participants, however, for each share s_i we now check with all other parties in $\overline{T}_i$ that they received the same value for the given share. If any inconsistencies are detected for a share, the parties broadcast a complaint. For any share where complaints were raised the dealer broadcasts the correct value of that share, and the parties accept the new value as the correct one. Note that an adversary is able to choose a new value for the share in this step, but the same effect could be achieved if the adversary had this new value as the original value of the share. If the dealer refuses to broadcast a share, the protocol is aborted, since it cannot guarantee correctness otherwise. To summarize:

VSS Share (Σ, $\mathcal{D}$)

1. Split secret s into k shares s_i: Select $k-1$ shares $s_1, \ldots, s_{k-1}$ randomly from the domain $\mathcal{D}$. Let $s_k = s - \sum_{i=1}^{k-1} s_i$
2. Send share s_i to all parties in $\overline{T}_i$
3. For each share s_i: All parties in $\overline{T}_i$ send share s_i to each other over a secure channel, to make sure they agree on the value.
   If any disagreements are detected, a complaint is broadcast.
4. The dealer broadcasts all shares for which complaints were raised, and the parties accepts these shares. If the dealer refuses to broadcast, the protocol is aborted.

The verification of share values introduces more attacks for the adversary. Examples of these include:

- *Illegal verify*: When verifying shares with other parties, a corrupted party tries to verify a share he is not qualified to know with a qualified party, in order to get the party to raise a complaint and trigger a broadcast, thereby obtaining the value of the share.
- *Illegal complaint*: A corrupted party complains about a share that he is not qualified to know, in order to get the dealer of the share to broadcast it.

To protect against illegal verify, a check will be made in step 3, to only verify shares with parties in the correct $\overline{T_i}$ set. To protect against illegal complaints, a check can be made in step 4 of the protocol, to only accept complaints for a share s_i by the parties in the set $\overline{T}_i$. In general, since the parties agree on the distribution set, a party can always check if another party is sending messages about a share that he is not qualified to know and ignore these messages.

4.2.2 VSS Reconstruct

Let us now take a look at the steps needed for reconstructing a secret. To perform a reconstruction, all parties firstly send all their shares in a bilateral fashion to all other parties. These shares then need to be checked to prevent attacks by the adversary. Recall that the shares sent by a given party only correspond to a subset of all the shares of the secret. As such, values of s_i are sent only by the parties in $\overline{T_i}$. Each of these values of s_i needs to be checked against each other in order to find a unique correct value for s_i, which corresponds exactly to the share value sent by the honest parties of $\overline{T_i}$. Finding this value is the act of reconstruction. After all the individual shares have been reconstructed, the secret is simply obtained by summing up the shares.

The reconstruction of the individual shares is performed as follows: If all values received for s_i agree, i.e. $v_j = v$ for all $j \in \overline{T_i}$, we can simply choose this value as the correct value of s_i. On the other hand, if differing values have been received for s_i, we need to identify which is the correct one. This is achieved by iterating over all subsets of the adversary structure, and seeing if the senders of the differing values are contained in one of the sets in the adversary structure. For each subset in the adversary structure A, take a copy of all the values received for s_i, and remove the shares sent by the parties in A from the copy. If all the remaining values agree, choose this as the correct value of s_i. This corresponds to $v_j = v$ for all $j \in \overline{T_i} - A$. In summary, the reconstruct protocol works as follows:

VSS Reconstruct (Σ, Δ)

1. All parties send all their shares (bilaterally) to all other parties.
2. Each party reconstructs (locally) each of the k shares $s_1, \ldots, s_k$ and adds them up to obtain the secret $s = s_1 + \cdots + s_k$.
   Reconstruction of share s_i (same for each party): Let v_j for $j \in \overline{T_i}$ be the value (for s_i) sent by party p_j. Take the (unique) value v such that there exists $A \in \Delta$ with $v_j = v$ for all $j \in \overline{T_i} - A$.

At a first glance at the protocol, it might seem overcomplicated to determine the correct share value in the way presented. One might be tempted to think that choosing the majority value, among the values received for a share s_i, is sufficient. This would indeed be the case if adhering to the threshold condition for an active adversary in an information-theoretic setting, as presented by [4] and [5], since this would mean that we were guaranteed an honest majority in each subset of the distribution set. But, since we are working with a general Σ and Δ, this is not the case. We will prove this by contradiction with an example that fulfills (2), yet does not does not yield the correct value upon taking the majority. The setting is as follows:

$$n = 7$$

$$\Sigma = \{\{1,2,3\},\{4\},\{5\},\{6\},\{7\}\}$$

$$\Delta = \Sigma$$

$$\overline{T_1} = \{4,5,6,7\}, \overline{T_2} = \{1,2,3,5,6,7\} \ldots$$

If we're looking to reconstruct s_2, we see that an honest majority cannot be achieved if the adversary subset of $A_1 = \{1,2,3\}$ all send an incorrect share value v_j that differs from the honest parties' value v_i.
To get a quick intuition of the way of selecting the correct share value as presented in the protocol, look again at the example presented above. If we remove A_1 from T_2 only one unique value remains. This value must be sent by honest parties, since the adversary can only corrupt a single subset of Δ at a time.

4.2.3 Proofs

Correctness
Correctness for **VSS** means that reconstructing a secret shared value results in each honest party having the same value and that if the dealer of the shares is honest, then the reconstructed secret at each party is the same as the input secret [1].

To argue for correctness, we first argue that the result of **VSS Share** is, that for each share s_i, all honest parties who know this share agree on its value, and that if the dealer was honest, then the value of s_i was the one the dealer sent. This is handled in the verification part of the protocol and ensured by the fact that honest parties will always raise complaints about differing shares, and all complaints must either be handled, or the protocol aborted. If any share is broadcast due to a complaint, the correctness of this part of the protocol follows from the correctness of the underlying broadcast protocol.

Next, we argue that if all honest parties that know a share s_i agree on the value for s_i, then **VSS Reconstruct** results in all honest parties having the secret that was initially shared. To argue this, we mainly need to argue that the reconstruction of the individual shares s_i is correct. Here, correctness can be ensured if condition (2) is satisfied. Condition (2) implies that when receiving a collection of different values for a share s_i, there is only one consistent explanation for what the correct value is. This is due to the fact that the inconsistencies will always correspond to exactly one set in the adversary structure. To see why this is the case, consider a setting where two different values v' and v'' could be considered correct values for the share s_i. Both of these values would have to have corresponding sets A' and A'' in the adversary structure to be able to explain the value of s_i. Consider then the set $P \setminus A' \cup A''$. This set cannot be qualified since otherwise it would uniquely determine the value of the share, which is not the case since the values v' and v''

differ. But the set cannot be ignorant either, as this would contradict (2). Since the set $P \setminus A' \cup A''$ cannot exist, the setting is impossible which means that there can only be one correct explanation for s_i.

Secrecy
The proof of secrecy for **VSS** relies partly on the proof of secrecy in the secret sharing scheme for passive adversaries. What remains to be argued is that the adversary cannot get any honest party to reveal shares that the corrupted parties are not qualified to know. Here, the broadcasting of shares in step 4 of **VSS Share** might seem like an issue, but this is not the case. For a complaint about a share to be raised, one of the corrupted parties must either be the dealer of that share, or be in the qualified set for that share. Hence the adversary already knows the value of that share anyway. For any share s_i that the corrupted parties are not qualified to know, the honest parties can always maintain secrecy by using the distribution set to avoid leaking information to those outside of $\overline{T_i}$.

4.3 Robust multiplication

Let us now consider the multiplication protocol for actively corrupted participants. The multiplication protocol described for the passive case fails with active corruption, as we delegated the computation of each $s_i t_j$ term to exactly one of the parties. This means that a corrupted party can provide an incorrect value of any $s_i t_j$ term that he is responsible for. To prevent this type of attack, we need to extend our protocol. Instead of a party being responsible for computing a subset of the partition, each party now computes all $s_i t_j$ terms he can, and then shares these values with the VSS Share protocol.

Since there can be more than one party computing a given term, we want to compare these different versions. To do this, we utilize the fact, that if two different versions of a term $s_i t_j$, $(s_i t_j)'$ and $(s_i t_j)''$ are split into k shares,

$$(s_i t_j)' = (s_i t_j)'_1 + \cdots + (s_i t_j)'_k$$
$$(s_i t_j)'' = (s_i t_j)''_1 + \cdots + (s_i t_j)''_k$$

it should hold that

$$(s_i t_j)' - (s_i t_j)'' = 0$$

and in turn

$$(s_i t_j)'_1 - (s_i t_j)''_1 + \cdots + (s_i t_j)'_k - (s_i t_j)''_k = 0$$

Therefore, for each secret shared term $s_i t_j$, each party computes the differences between the versions of the term, for the shares he is qualified to know, and then computes the sum of these share differences utilizing VSS Reconstruct, to determine whether each difference equals zero. If all differences are zero, the different versions of the term are equal and the parties can use any arbitrary version as the correct sharing. If any difference is not zero, s_i and t_j are reconstructed, and $s_i t_j$ is shared in some default manner. Any way of sharing works here, since all parties know the values s_i and t_j due to the reconstruction. In summary we have:

Robust Multiplication Protocol (Σ)

Precondition: Two values $s = \sum_{i=1}^{k} s_i$ and $t = \sum_{i=1}^{k} t_j$ are shared using **VSS Share**

Postcondition: st is shared using **VSS Share**

1. Each party p_m computes all terms $s_i t_j$ he can (i.e. those for which $m \in \overline{T_i} \cap \overline{T_j}$) and shares them using **VSS Share**.
2. For each pair (i, j), let $(p_{m_1}, \ldots, p_{m_r})$ be the ordered list of the parties who computed $s_i t_j$ in step 1.
   All n parties compute and **reconstruct** the differences between the shares received from p_{m_1} and the shares received from p_{m_i} for $i = 2, \ldots, r$.
3. If all these sums are 0, then the sharing by p_{m_1} is chosen as the sharing of $s_i t_j$.
   Otherwise, s_i and t_j are **reconstructed** and the k-out-of-k sharing for the term $s_i t_j$ is defined (arbitrarily) as the list $(s_i t_j, 0, \ldots, 0)$ of shares.
4. All n parties locally compute the sum of their shares of all terms $s_i t_j$, resulting in a sharing of st.

Correctness
As with the passive multiplication protocol, correctness means that the postcondition is upheld. This is mostly guaranteed by the correctness of VSS. Besides breaking the correctness of these protocols, the only attempt the adversary has at breaking correctness is getting complete control of one (or more) of the terms, as this would allow the adversary to assign a wrong value to it, and get it accepted without reconstruction in step 2 and 3. Since each term is computed by a set in the complement of $\Sigma \sqcup \Sigma$, the adversary cannot be allowed to control any of these sets completely. This is ensured exactly when condition (3) is satisfied.

Secrecy
For step 1, the secrecy of the protocol relies on the secrecy of VSS. For step 2 and 3, note that reconstruction can only happen if an adversary has computed a wrong value for $s_i t_j$, meaning that the adversary is already in possession of s_i and t_j. Therefore, the reconstruction does not violate secrecy. Step 4 is done locally and can therefore not violate secrecy.

5 Implementation: Active participants

5.1 Structure and approach

The MPC protocol for active adversaries, which we present, follows the same pattern as the protocol for passive adversaries. We still think of an MPC as an initial sharing phase, followed by a series of addition and multiplication instructions, and lastly a reconstruction of the result of the MPC. Thus, the main differences between the active and the passive case is in the secret sharing, multiplication and reconstruction protocols.

Note that our implementation is in a local setting, and only models the distributed setting in which an actual MPC would take place. This means that we have abstracted away some of the aspects that one would have to consider in a distributed setting. For example, we have chosen not to implement a signature scheme, which the parties would need to have to prevent spoofing attacks by the adversary. Instead, we simply assume that a signature scheme is in place, and as such, in our model, a corrupt party does not have the capability to put anything but their own party number in the sender field of a message. We have also chosen to abstract away the broadcast channel that the parties need to ensure they have the same values. Instead, broadcasting in our implementation simply means sending to all other parties. To model the effect that broadcasting would have in practice, we do not let the adversary send different values to different parties in the broadcasting parts of the protocol. Furthermore, an adversary refusing to send a message to some party

is modelled by sending a message regardless, annotated "Refuse". Likewise, an honest party cannot crash and messages cannot be lost.

These abstractions serve several purposes. They allow us to avoid using timeouts, for streamlined execution of the protocol and tests. They also serve to make the theory underlying the protocols clearer, since we can avoid code that does not have anything directly to do with the theory of MPC. Lastly they mean that we do not have to commit ourselves to e.g. a public key infrastructure for signatures.

Note that the abstractions do not result in any loss of generality. The creation and verification of signatures could be added to any send/receive methods, and the round based model needed for broadcasting as well as the broadcasting protocol itself could be added instead of our approach of using waitgroups and goroutines. For the last part to be true, the outermost function MPCProtocolActive() must not provide the parties with information that they would not have in a round based protocol, where there is no trusted party. MPCProtocolActive() adheres to this condition as it solely does two things: Tells the players when they can proceed in the protocol, which would be handled by the rounds in the protocol, and provides the players with information that could also be obtained at the start of the protocol or during the protocol using broadcast. For example, in our implementation of **VSS Share**, the parties need to know the total number of complaints to wait for. This is given to the parties by MPCProtocolActive() in our implementation, but could also be achieved by each party broadcasting their number of complaints. Similarly, in some parts of the implementation, MPCProtocolActive() reads exitcodes set by each party, and uses this information to branch on some functions. The same effect could be achieved if the parties broadcasted these exit codes and branched individually.

For the testing of this part of the implementation we chose to simulate an adversary in order to verify that our implementation was safe against various attacks. This was done in the file adversary.go by creating a copy of the main function, MPCProtocolForTests(), where some of the parties deviate from the protocol, usually by injecting illegal messages in the system. For instance, to test that complaints are raised and handled properly we corrupted one party by sending different values of some share from this party.

5.2 VSS Share

In our implementation, VSS Share is executed in 5 phases:

- Distribute: Create shares from a party's secret and share these using the basic secret-sharing scheme.
- Verify: Send out each party's version of their shares for verification and generate potential complaints.
- Broadcast complaints: Broadcast all generated complaints.
- Resolve complaints: Resolve all broadcast complaints.
- Check for aborts: Check if any party signaled for abort as a result of the Resolve complaints phase. If so, terminate the MPC protocol for all parties.

Of these, the first 4 phases run in the usual fashion of sending some message and waiting for everyone to receive it. Note that a complaint is modelled as a Message of type "Complaint". Since messages cannot be lost and we can always determine how many messages each party is to receive during a given phase, we can always ensure that a party is in the right state at the start of each phase with this implementation. This is also the reason behind our decision to have the complaints broadcast and resolved in separate phases, rather than doing these phases concurrently. If we kept to the 3 steps of **VSS Share** as described in the theory section, it would be rather difficult to deduce when a party has received all complaints in step 2, since this depends on every party being done with their respective verifications. By 'resolving a complaint' we refer to step 4 of **VSS Share**, where the dealer of the share that raised the complaint either broadcasts the share or broadcasts a "Refuse" message, as described earlier. As such, a complaint message is annotated with the index of the complained-about share and the dealer of that share.

The last phase "Check for aborts" iterates through the parties to check if any of them set their exit code to Abort during the Resolve complaints phase, upon having received a Refuse message. If that is the case, the protocol aborts, otherwise, it proceeds normally.

The attacks presented in the active theory section is handled as follows: For *Illegal Verify*, when a party receives a verify message in receiveVerifications(), that is, a share that the party must compare to his own version of the share, the party must check to make sure that the sender of the value is in the qualified set for that share. Similarly, for *Illegal Complaint*, when receiving a complaint in receiveComplaints() the receiving party must check to make sure that the sender of the complaint is in the qualified set for the share that he is raising a complaint about.

It should be noted that **VSS Share** appears in two versions in our implementation, specifically VSSShareInit() and VSSShareMult(), both of which adhere to the above phases and are complete implementations of the protocol. We chose to make two versions because of the vast amount of shares, that need to be handled during the multiplication protocol. As such, we implemented a data structure called a term matrix [2], utilized by VSSShareMult() to make it more efficient, by sending out matrices of shares, rather than individual share slices, which is the case for VSSShareInit(). VSSShareInit() handles the sharing of each party's input secret, whereas VSSShareMult() handles the sharing during the Multiplication protocol.

5.3 VSS Reconstruct

For reconstruction, we have two different functions corresponding to the two scenarios for reconstruction in the Multiplication protocol; One for reconstruction of a factor reconstructFactor() and a more general one, reconstruct(), used both for reconstruction of differences slices in the multiplication protocol and reconstruction of the result of the MPC. These both handle step 1 of **VSS Reconstruct**.

The main workhorse function is completeShareSlice(), which handles the actual reconstruction, as described by step 2 of **VSS Reconstruct**. The function takes as input the shares sent in step 1, represented as a matrix of shares, with dimensions $r \times k$, where r is the number of share slices received, such that each column c_i holds the values of share s_i. It then iterates column-wise through the matrix and creates a map, map[party number $\Longrightarrow$ share value], filled with key/value-pairs of senders of the given entry and the value at that entry. It then uses the map to deduce which set in Δ is responsible for the differing values, by iteratively removing each set, A, in Δ from the map and checking that all values match (the map is reset between each iteration). When the correct set A is found, only the unique value v is left and is chosen as the correct value of s_i. If no differing values are present, the last entry in the map is chosen for s_i, arbitrarily. Upon having worked through all columns, the function yields the slice of correct shares, corresponding to the right-hand side of $s = s_1 + \cdots + s_k$.

After completeShareSlice(), the parties all have the same share slice. The only step missing is computing the sum of the shares to obtain the secret. In contrast with the theoretical protocol, this step is done outside of the actual reconstruct functions.

5.4 Robust Multiplication

The main issue with implementing the multiplication protocol was the sheer amount of values that needs to be passed around to ensure correctness. Step 1 of the protocol, the VSS sharing of the terms $s_i t_j$, involves sending $O(n^2k^2)$ share slices, since each of the n parties need to send shares of all of the k^2 possible terms $s_i t_j$ they can compute, to all of the other parties. The verification phase of the VSS sharing as well as step 2 of the multiplication protocol entail similar amounts of values to be sent.

To improve efficiency, we decided to bundle together the values that one party needs to send to another party, in a data structure called a term matrix. A term matrix is conceptually a $k \times k$ matrix with an entry for each possible pair $\{(i,j) : 1 \leq i,j \leq k\}$, and each entry consisting of a share slice. In practice, this is

[2] See more under Robust Multiplication

implemented as a doubly nested slice.

In the Distribute phase of VSS share for the terms, distributeMultShares(), each party must first compute a temporary term matrix, filling out each entry (i, j) it can according to the distribution set, that is, for a party p_m, the entries (i, j) where $m \in \overline{T_i} \cap \overline{T_j}$. The entries (i, j) are filled with k shares of the term $s_i t_j$. Each party then iterates through his term matrix and fills out n new term matrices, one for each party. These new term matrices have values in the same entries as the temporary term matrix, but are only filled with the shares that the player, which is to receive the term matrix, is qualified to know. After the Distribute phase of VSS share, each party has n term matrices and can proceed with the rest of the phases of VSS in the same manner as in the initial share sending.

After step 1 of the multiplication protocol, the players need to compare the different versions of each of the terms. This is handled in step 2, where each term $s_i t_j$ is handled individually in an iteration.

For each term, computeDifferences() is called. In computeDifferences(), each party figures out the r parties, $p_{m_1}, \ldots, p_{m_r}$, which computed that term, using the distribution set, and finds their shares of the term in the term matrices. For $i = 2, \ldots, r$ they then subtract the shares of p_{m_i} from p_{m_1} to create partially filled $r - 1$ difference slices. After that, they reconstruct the difference slices with the rest of the parties, to complete the slices. Finally, in checkForReconstructs() the parties compute the sum of each difference slice, while using modulo to stay within the domain.

Step 3 and 4 of multiplication protocol follow the theoretical protocol closely, the only real difference being that the sharing of each term is added to the result before the next term is considered, instead of the entire sum being computed at the end.

6 Benchmarking

6.1 Results of performance tests

To benchmark the efficiency of the presented MPC protocols, we conducted a range of performance tests. These were conducted by using the built in benchmarking features of Go. Go's benchmarking works by running a given function, and measuring the time it takes to complete the function. It runs said function enough times to get consistent results for the runtime, and runs for at least one second. We run this benchmarking protocol five times for each test we conduct, and take the average of these values for our final data point, as to minimize measurement uncertainty. The actual time spent on the performance tests is dependent on the hardware that executes them and any background noise from other processes. It is therefore more relevant to discuss trends in the scaling of the protocols, as well as comparing performance of protocols, rather than discussing actual numbers for the runtime. The benchmarks were run on a local machine with a Darwin OS, arm64 architecture and an Apple M1 CPU that used 8 cores. For further information, see [7] and our code in appendix A.1.

When conducting these tests, we use a standard set of inputs for the protocol, where one variable is varied for each specific performance test. We do this to highlight how a particular variable affects the running time of the protocols. The standard inputs are:

- Number of parties: 5
- Secrecy structure (Passive): The set of all maximal subsets of the parties satisfying the corruption threshold $t < n/2$
- Secrecy structure (Active): The set of all maximal subsets of the parties satisfying the corruption threshold $t < n/3$
- Secret: Randomly chosen in the range [0-99]
- Number of instructions: 3 (computing $(a \cdot b) + (c \cdot d)$)

The performance tests that we conduct are:

- Running time for the passive vs the active protocol as a function of number of parties
- Running time for active protocol as a function of the number of subsets in the secrecy structures
- Running time for the passive vs the active protocol as a function of number of instructions
- Running time of the Addition vs. the Multiplication protocol as a function of number of instructions

For the passive cases, the minimal secrecy structure is the set of all subsets of size 1. No adversary structure is used in the passive cases. For the active cases, the minimal secrecy structure was constructed similarly, also yielding subsets of size 1. The adversary structure is equal to the corresponding secrecy structure in the active cases.

All data is presented in graphs, with an added trend line, to indicate how the running times scale. The coefficient of determination R^2 is also included, to show how well the trend line fits the data.

6.1.1 Passive vs Active: Varying number of parties

For the number of parties, n, we used a range of 4 to 10, since we needed $n > 3t$ for the active secrecy structure and anything beyond 10 parties with a maximal secrecy structure gives an exponential explosion in terms of running time, which meant that the tests would take too long to run. Since the running times increase exponentially we use a logarithmic scale on the y axis to portray the data.

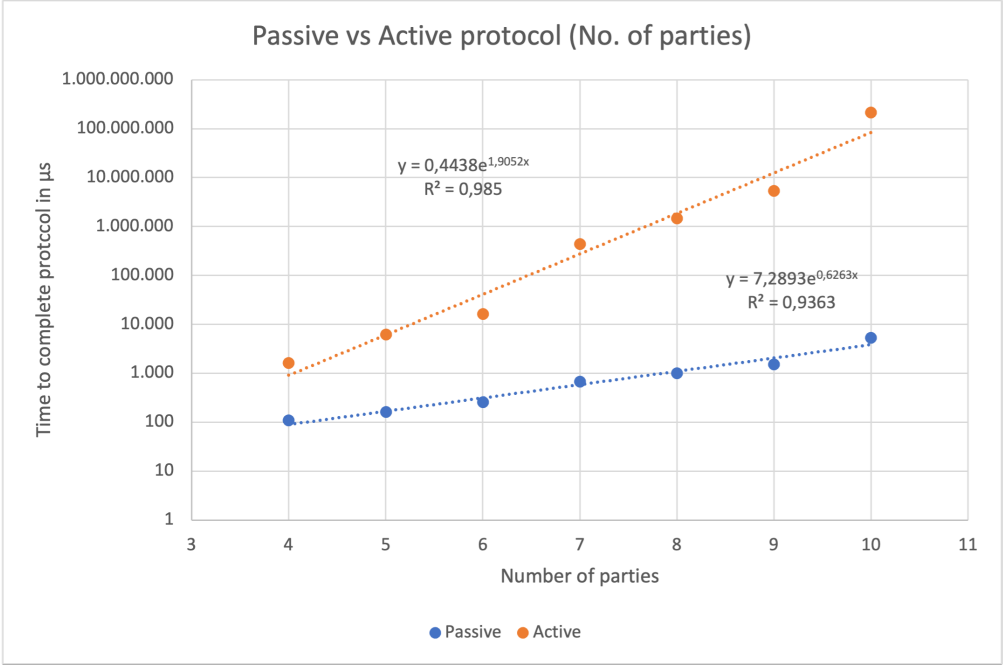


Figure 1: Time to complete protocol for the standard input as a function of number of parties. A base-10 logarithmic scale is used on the y-axis.

It can be seen in figure 1 that the running time for both the MPC protocols increases dramatically when the number of parties increases. Both protocols show an exponential growth rate, which indicates that they do not scale well when the number of parties participating in the protocol becomes too large. Furthermore, the growth rate for the protocol for active corruption is seen to be much higher than that of the protocol for passive corruption.

6.1.2 Varying size of secrecy structure

For this test, we only consider the active protocol, as this is enough to illustrate the correlation between running time and size of the secrecy structure. This test was performed for 10 parties rather than 5, as to be able to create more subsets in the secrecy structure. Our range of data for this chart starts with a secrecy structure of size 10. We then increased the size by increments of ten, ending up at a secrecy structure with 100 combinations, that still adhere to conditions (2) and (3).

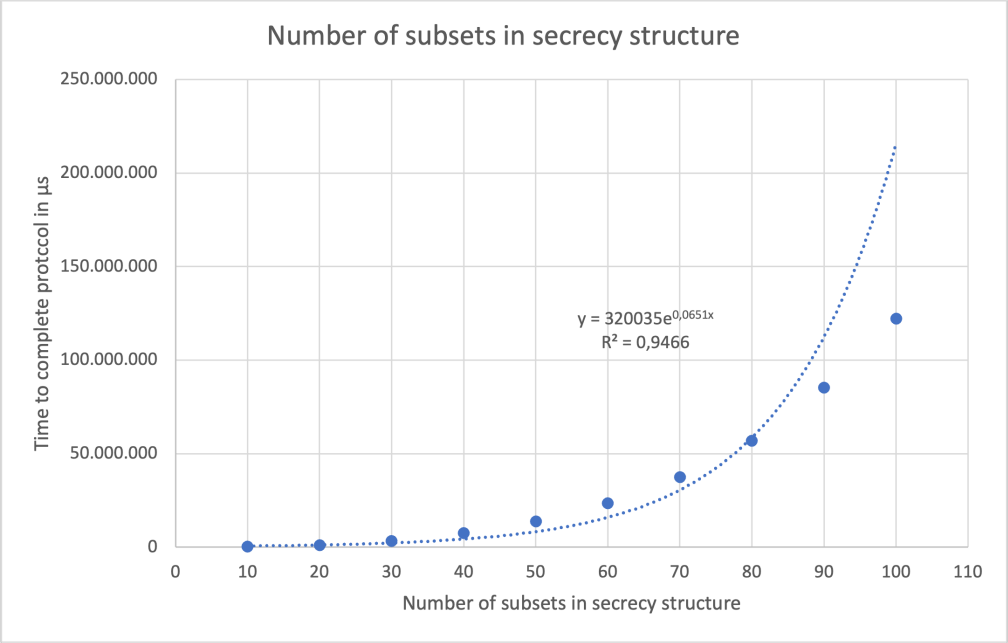


Figure 2: Time to complete protocol for the standard input (apart from number of parties) as a function of the number of subsets in the secrecy structure.

It can be seen, that when increasing the number of subsets in the secrecy structure, the time spent to complete the protocol increases drastically.

6.1.3 Passive vs. Active: Varying number of instructions

For this test, the number of instructions for each protocol to compute was varied, following the pattern $(a \cdot b) + (a \cdot b) + ... + (a \cdot b)$. We remind the reader that $a \cdot b$ is one instruction and adding $a \cdot b$ to the result is another instruction. Note that for presentational purposes, the last addition instruction is excluded, such that the final $a \cdot b$ is computed but not added to the result.

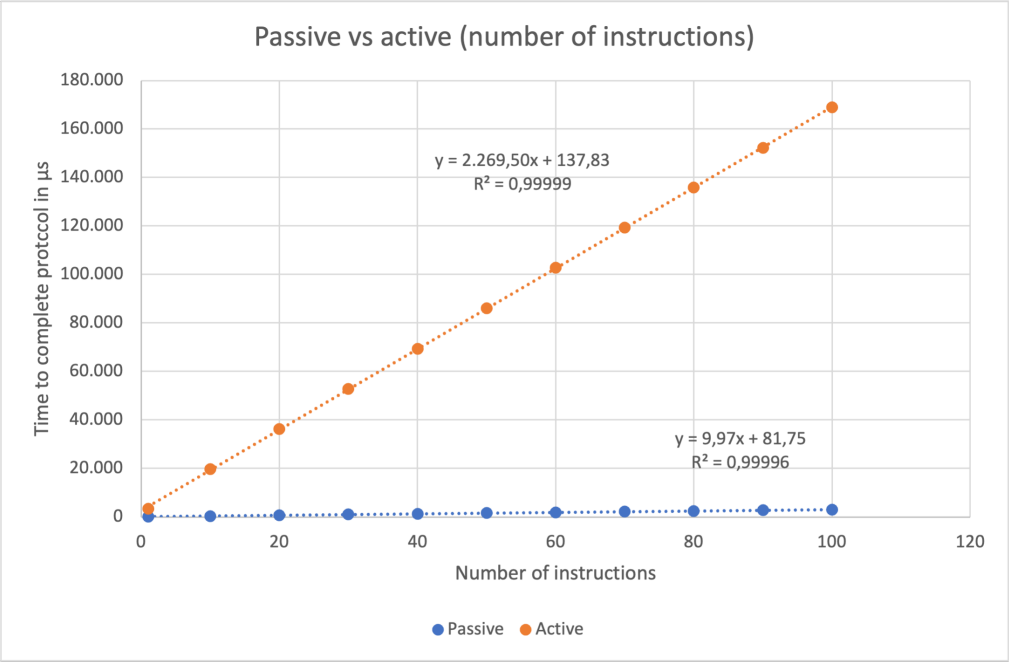


Figure 3: Time to complete protocol for the standard input as a function of number of instructions.

It can be seen that the time to complete both protocols, increases in a linear manner, when increasing the number of instructions the protocol has to complete. It is worth noting, that even though both scale linearly, the active protocol, scales much more drastically, than the passive protocol.

6.1.4 Addition vs. Multiplication (Active)

For this test, note that we again only examine the active protocol following the same reasoning given for the performance test for number of subsets in the secrecy structure. The expressions for the increasing instructions follow the pattern: $(a \diamond b) \diamond a \ldots \diamond a$ for $\diamond = \{+, \cdot\}$.

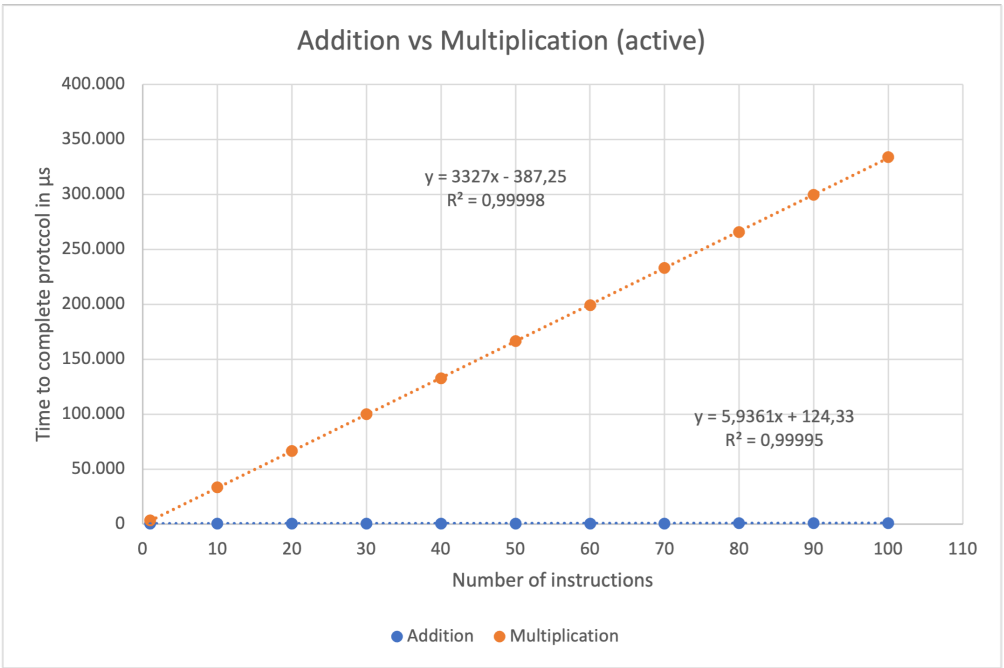


Figure 4: Time to complete protocol as a function of number of instructions, respectively for active addition and active multiplication.

We quickly notice that computing even a large amount of additions take only a fraction of the time it takes to complete just 10 multiplications. Each addition instruction adds only about 6 microseconds of computation time, whereas a multiplication instruction takes more than 500 times longer. The main take away here is that the multiplication protocol significantly outweighs addition in terms of computation time, to a degree where the influence of additional addition instructions is negligible.

6.1.5 Interpretation of results

From the performance tests, we can conclude that the running times of our protocols is heavily influenced by the number of parties and the size of the secrecy structure. This is evident from the graphs presented, as both increasing the number of parties, as well as the size of the secrecy structure, led to an exponential growth in the time taken to complete the protocol. This is primarily due to the way the secret sharing scheme is defined, and the way the protocols depend on k, the size of the secrecy structure.

As the distribution set has the same size as the secrecy structure, increasing the size of the secrecy structure leads to a larger distribution set. The increase in the size of the distribution set especially has an impact on the running time for the active setting, as the VSS scheme entails redundant sharing since the parties have to verify shares with other parties, each time they receive them. This occurs both in the initial secret sharing, but in particular when performing the multiplication protocol, as the number of terms is k^2 and all terms need to be shared and verified.

Similarly, when the number of parties is increased, the size of the maximal secrecy structure increases

exponentially, which gives a corresponding exponential growth in running time for the same reasons as above. Thus, the number of parties in the protocol mostly have an influence of the worst case running time of the protocol, namely the case where the secrecy structure is maximal.

Another factor that influenced the running time of the protocol was the number of instructions the protocol has to complete. The running time of the protocol, as a function of number of instructions progressed linearly for both addition and multiplication. This was to be expected as we are simply repeating a computation a number of times, and each instruction computed has the same number of parties participating. As noted earlier, the time cost for performing multiplications is much greater than that of additions. This was also as expected, as the active protocol involves much more sharing and verifying, as we have to share and verify each term the parties compute.

Lastly, the passive protocol is significantly faster than the active protocol, which can be attributed to the increased number of messages that needs to be sent to keep the different shares in the protocol consistent.

7 Future work

The main disadvantage of our protocols is the poor scalability when the size of the secrecy structure increases. Thus, a natural goal for future work would be to improve the scalability of the protocols. In this section we will present alternative approaches that would decrease the running time of the protocols, at the cost of making the protocols more complicated. We will focus on the active MPC protocols, since they have the longest running times, as shown by our performance tests.

Note that our use of a simulated network means that communication between the parties is really fast. In a real-world distributed setting, communication between the parties would be one of the biggest factors in regards to running times, as the time spent on various network delays could easily outweigh the local computation time needed at the parties. As such the point of some of the alternative approaches we present will be to reduce the amount of communication needed in the protocols.

For secret sharing, one of the main factors of the running time of the protocol is the size of the secrecy structure $k = |\Sigma|$, as shown by the performance tests of section 6.1.1 and 6.1.2. Therefore, one way to increase efficiency would be to consider other approaches for secret sharing. As presented by Shamir [8], it is possible to secret share using only n shares, where n is number of parties in the protocol. Reducing the number of shares from k, which is potentially exponential in n, to n would significantly increase the efficiency of both secret sharing and multiplication, especially the latter, since k^2 terms need to be handled. Of course, our proofs for security and correctness depend on the secrecy structure being used in the protocols, so they would no longer be useful.

Different approaches could also be considered for multiplication since our protocol requires a lot of communication between the parties. In a setting with a real network, this would further increase the running time of our protocol, which is already mostly dependent on the multiplication protocol, as shown by our performance test in section 6.1.4. We first present proposals to improve on our own implementation without changing the core theoretical protocol, then point to a completely different approach which focuses on reducing the communication complexity.

The main part of the communication in the multiplication protocol happens in the iteration over the terms. For each term, the reconstruction protocol is first run $r-1$ times where r is the number of parties that know the term. Using the worst case secrecy structure, which is the one defined by the corruption threshold $t = \lceil \frac{n}{3} \rceil - 1$, r is the size of the intersection of two sets in the distribution set. Therefore, we choose to upperbound r by the size of the individual sets in the distribution set, $n-t$. Each reconstruction involves sending n^2 messages, since all parties need to send versions of the share to be reconstructed to each other. Thus this part of the protocol has a communication complexity of $O(r-1)n^2 = O((n-t-1)n^2) = O(n^3 - tn^2)$. Afterwards, if the factors of the term need to be reconstructed, the reconstruction protocol is run two additional

times. Here, only the parties that know the factors need to send their value out. Since each factor is known by $n-t$ parties, and each of the parties need to send to all other parties, this part of the protocol has a communication complexity of $O(2(n-t)n = O(n^2-tn)$, which is subsumed by $O(n^3-tn^2)$. Since there are k^2 terms to be handled, the communication complexity of the entire iteration is $O(k^2(n^3-tn^2)) = O(k^2n^3-tk^2n^2)$.

To reduce the amount of communication, each party could instead gather all the shares of all the differences of all the terms in some data structure and send this data structure in a single packet, provided the data can fit in a single package. Afterwards each party could send all the factors that needs reconstruction in a single package. This would drastically reduce the communication complexity, since each party would only need to send 2 messages to each other party, resulting in a complexity, for this step of the protocol, of $O(n^2)$.

However, the communication complexity of the entire multiplication protocol would still be at least $O(n^3)$ since the VSS Sharing phase of the multiplication protocol has a complexity of at least $O(n^3)$. To see this, note that even with the term matrices, each party needs to send shares to each other party. This means that the adversary can raise at least one complaint for each party, by sending out incorrect values during the verification phase. In response to these complaints each party must then make at least one broadcast, which has a communication complexity of $O(n^2)$ [9].

Therefore, we cannot bring the communication complexity of the multiplication protocol lower than $O(n^3)$, without making the protocol rely less on broadcasting. One way to do this would be to use the multiplication protocol presented by Hirt and Maurer [9]. Here, a more involved preparatory phase is used to ensure a communication complexity of $O(n^2)$. However the reduced complexity comes at the cost of the simplicity that the protocol presented in this paper achieves.

8 Conclusion

In this paper we have presented the theory behind, and implementation of a general, perfectly information-theoretically secure MPC protocol. Since the theoretical protocols we have presented use general adversary structures and domains, they can be adapted to a multitude of settings, including but not limited to classic threshold based adversary models. Due to their simplicity, they can be used both as a basis from which to extend the theory of multi-party computation, and to gain an introductory understanding of the subject.

Besides the theoretical protocols, we have presented a working implementation of MPC intended for a small number of participants, which can be used to compute any function in a secure manner, without having to trust any specific party. Which specific MPC protocol to use, will depend on the given attack model, the adversary being in either a passive or active setting. We should emphasize that this requires the given function to be translated into a series of additions and multiplications. Furthermore, for distributed use cases, users of our implementation will have to refactor the code to use real network communication.

In the paper we have discussed various issues regarding this variant of MPC. One of the issues was how to handle the main drawback of the protocols, namely the poor scalability. We have presented some alternatives to the protocols that handle large amounts of parties better, but besides that, concrete real world applications with clearly defined adversary settings could use the presented protocols as inspiration, or as a baseline from which scalable solutions could be implemented.

9 Acknowledgements

We would like to give a special thanks to our advisor, Luisa, for her invaluable feedback and helpful discussions during the writing of this paper.

A Appendix

A.1 Code

The codebase for our project can be found in the github repository located at:
https://github.com/limesacul/secure-MPC

References

[1] Ueli Maurer. "Secure multi-party computation made simple". In: *Discrete Applied Mathematics* 154.2 (2006). Coding and Cryptography, pp. 370–381. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2005.03.020. URL: https://www.sciencedirect.com/science/article/pii/S0166218X05002428.

[2] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. eng. West Nyack: Cambridge University Press, 2015, pp. 7–18. ISBN: 9781107043053.

[3] Martin Hirt and Ueli Maurer. "Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract)". eng. In: *Proceedings of the sixteenth annual ACM symposium on principles of distributed computing*. PODC '97. ACM, 1997, pp. 25–34. ISBN: 9780897919524.

[4] A. Wigderson M. Ben-Or S. Goldwasser. "Completeness theorems for non-cryptographic fault-tolerant distributed computation". eng. In: *Proceedings of the 20th ACM Symposium on the Theory of Computing (STOC)*. ACM, 1988, pp. 1–10.

[5] I. Damgård D. Chaum C. Crépeau. "Multi-party unconditionally secure protocols (extended abstract)". eng. In: *Proceedings of the 20th ACM Symposium on the Theory of Computing (STOC)*. ACM, 1988, pp. 11–19.

[6] Henrik Baerbak Christensen. *Flexible, reliable software : using patterns and agile development*. eng. Chapman & Hall/CRC textbooks in computing. Boca Raton: Chapman & Hall/CRC, 2010. ISBN: 9781420093629.

[7] Dave Cheney. *How to write benchmarks in Go*. URL: https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go. (accessed: 16.05.2022).

[8] Adi Shamir. "How to Share a Secret". In: *Commun. ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176.

[9] Martin Hirt and Ueli Maurer. "Robustness for Free in Unconditional Multi-party Computation". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 101–118. ISBN: 978-3-540-44647-7.