MASCOT Private Meeting Scheduler

Jonathan Gold
University of Waterloo
jsgold@uwaterloo.ca

Parsa Salimi
University of Waterloo
p2salimi@uwaterloo.ca

ACM Reference Format:
Jonathan Gold and Parsa Salimi. 2022. MASCOT Private Meeting Scheduler. In *Proceedings of CO 789 Final Paper (CO789).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Scheduling meetings is a daily occurrence for many people around the world. This is a task crucial to our daily lives and in 2022 is often performed in a digital setting. Most people hold digital calendars which can be accessed by coworkers, colleagues or friends. This makes scheduling meetings or events very straightforward. If someone wants an alternative to this and does not want to share their entire schedule publicly (or to large groups), other scheduling applications exist such as Calendly, Doodle or When2Meet. However, these options still involve sharing large portions of ones schedule with those who they want to meet with.

All of these ways of scheduling a meeting involve at least one member sharing their schedule with the rest of the meeting members. These methods will be described in more detail shortly, including the exploration of how much unnecessary personal information meeting members are revealing to each other (or publicly). We believe that there should be a method to schedule meetings in a way where the members of the meeting can reveal as little of their schedule as possible.

Microsoft Outlook's calendar feature is very often used to schedule meetings in the workplace. To make scheduling meetings easy, it is very common in many workplace environments to keep an updated calendar which is visible to other coworkers, whether it be on the same team or the entire organization. If someone wants to schedule a meeting, they will line up all other meeting members' schedules and try to find a time that works for everyone. This is a very good system from an efficiency perspective. Being able to look at everyone's schedule makes finding a time to meet as straightforward as possible. However, this method is very problematic from a privacy perspective as anyone involved in the scheduling of the meeting can see everyone else's schedule. The efficiency vs. privacy trade-off has a clear preference of efficiency in this model. Google Calendar, Microsoft Teams and other popular workplace calendar software offer similar approaches.

CO789, Waterloo, ON,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Outside of a workplace setting there are various applications, such as Calendly, Doodle and When2Meet (among others), which can be used to schedule meetings, all using differing methods.

Calendly offers many different types of meeting scheduling, including one on one meetings, round robin team meetings, group meetings and collective meetings. All of these rely on the host or hosts of the meeting making their schedules available to all meeting invitees. The meeting invitees will then pick a time that works for them. This offers a different kind of privacy from the workplace setting as the workplace setting has all members' calendars available, where Calendly's approach has only the hosts schedule being public. While revealing less, this still has a major privacy concern for the host of the meeting as their schedule is fully revealed to invitees.

Doodle offers both the workplace option as described above and Calendly's one on one meeting method as features on their service. Additionally, Doodle offers group polling for group meeting slots, which contains very similar privacy concerns, as users are revealing their available times to the rest of the group. In polls, nothing is hidden from other users, so it provides the same privacy as the workplace examples above.

When2Meet involves the host setting their schedule and sending a link to all other participants. Each participant enters their name and schedule. The host ultimately picks a time based on an overlay of all users' schedule. The added benefit to this method from a privacy perspective is that the only identifying feature for the participant is their name, which they can anonymize if they choose to. However, the schedule is still available for all to see, and it may be possible (or easy) to tie that schedule to one of the members.

This leads us to trying to find a new approach to meeting scheduling. One in which users' schedules can remain private and limited information about their schedule, is revealed.

Multiparty computation (MPC) is a key cryptographic problem [3], which has advanced significantly in the past 10 years. MPC involves n parties who each hold a private input $x_1, x_2, ..., x_n$. As a group, they would like to compute a function $f(x_1, x_2, ..., x_n)$ on these inputs. The idea behind MPC is that this computation process should be private in the face of adversaries. We would like to be able to perform the computation without others being able to learn anything about the inputs. MPC and specifically MASCOT, an MPC protocol, will be explored in greater detail in section 2. MPC can be used to solve any arithmetic circuit, so if we can create an arithmetic circuit to schedule meetings, we will be able to schedule meetings in a private manner. In section 2, we begin by exploring the relevant background information for our Private meeting scheduler. We will provide detail on MASCOT[5], the MPC protocol used for our function. Next we provide details on MP-SPDZ[4], the repository framework on which our implementation is based on. Additionally, we explore the relevant networking and communication involved in the scheduling.

1.1 Contributions

We have 3 main contributions as outlined in the paper. The first contribution is formalization of the problem at hand, along with the function (or functions) to solve this formalized problem. The second contribution is a security analysis of the function we describe. The final contribution is an implementation of the protocol [9], alongside an experimental and theoretical discussion on the efficiency.

First we formalized the problem at hand. The problem addresses how clients can schedule a meeting while revealing as little of their schedules to the other meeting members as possible. We used MPC and specifically the MASCOT protocol as the framework. Our main contribution here is the creation of such a function, which we can implement in an MPC framework, that can find an optimal meeting time, given users schedules. This function will take in users schedules in the form of a binary vector as inputs and will output the optimal meeting time, the number of members able to attend this meeting, and a confirmation on the validity of all other users' inputs. One major concern with this, is how to approach the question of how we define the optimal meeting time. This is addressed in section 3, where we present multiple definitions of optimal in regards to this problem, and how we go about solving them. There are two main definitions explored, along with their full solutions, as well as a brief discussion on other possible definitions.

Now that we have the function (or functions), our next major contribution is performing a security analysis on this function. We begin by defining our level of security and providing a security definition, in terms of both privacy and correctness. Privacy and correctness are proven separately.

The final contribution in this paper is the implementation and subsequent efficiency discussion on the protocol. The function is implemented into the MP-SPDZ [4] GitHub repository framework, which provides the necessary MPC background to make the implementation of a function possible. The base version of the function and one of the alternative versions of the function were implemented. We created a working demonstration of the function in use, being accessed by multiple clients successfully. With our implementation, we tested the runtime under differing conditions. We altered the number of clients to see how this affected runtime. We also altered the length of the vectors inputted by the clients, and determined the affect this has on the runtime. Additionally, we briefly explore why we see the effects that we see. Finally, we test the runtime of the alternative version of the function, to see if there are any noticeable changes.

1.2 Related Work

Above, we have described different methods for scheduling meetings using various software. None of these options however consider privacy as a major component of the structure of the algorithms or processes.

One solution to the private meeting scheduling problem exists from Silaghi in [10]. The problem as specified here involves each party having a set of meeting times which work for them, a set of locations of where they would like to meet, and a set of constraints. Their solution involves searching the entire space for all possible meeting times and locations that fit all participants needs. Finally, there is a random selection among the combinations which work, to pick the winning time and location. The paper outlines how to do this in the context of MPC. However, this is an NP-complete problem, which we believe is unnecessarily complex for the majority of day to day meeting scheduling. We believe our solution to be much more efficient and similar in functionality to the types of applications most often used for the task in the real world.

Additionally, in [7], as part of their multiparty homomorphic encryption library, produce a private meeting scheduler as a demonstration. They take a polling approach to meeting scheduling. The creator of the meeting sends a poll to the server. Each of the members of the meeting log in and submit their schedules to the server. Upon completion, the server will send the intersection of available meeting times to the creator of the poll. Their assumptions are that the clients do not cheat with their inputs and the adversaries are honest but curious. Additionally, they assume that the creator of the meeting does not collude with the server.

This approach in [7] can almost serve as a basis for our approach with our meeting scheduler, with some key differences. First, we want the results of the protocol to be a proposed meeting time and not the intersection of points where all members are available. We feel that this reveals too much information. Secondly, we want the results to be revealed to all participants and not only the creator of the meeting as this provides more transparency and trustworthiness. Thirdly, our protocol should be secure against malicious adversaries. Next, we would like the use of multiple servers, to protect in the face of corrupt servers. Finally, we will not assume that clients do not cheat with their inputs.

2 BACKGROUND

The protocol designed and as described in section 3 has been implemented into a working program. This section describes the relevant background information required for the implementation of the protocol. The section begins in 2.1 by providing detail on MASCOT, the MPC protocol used to implement our function. MASCOT is a state of the art MPC protocol, which was used as the backing for the computation. Section 2.2 describes MP-SPDZ, the repository used for the implementation of the MPC protocol and our function. Finally, section 2.3 provides detail on the networking and communication required between parties, in order for the MPC to run.

2.1 MASCOT

The protocol used for the secure MPC of our private meeting scheduler function is MASCOT [5]. MASCOT is a protocol for secure MPC of arithmetic circuits over a finite field. It provides malicious security against a dishonest majority. This provides MASCOT with the strongest notion of security available. It does provide correctness and privacy in the strongest sense, but it sacrifices fairness and security against aborts.

MASCOT is based originally on the SPDZ protocol[3], as both protocols use a two phase approach. The first phase is the preprocessing phase, where the necessary overhead for the computation of the function takes place. The main piece of the preprocessing phase is the creation of multiplicative triples, used later in phase 2 to assist with multiplication. The second phase is the online phase.

The goal of these protocols is to have the preprocessing phase be more expensive, so that the online phase is cheap.

Typically, MPC protocols that contain such a such a strong notion of security require very expensive arithmetic operations, particularly multiplication. While SPDZ was revolutionary at the time, it is no exception to this. The SPDZ protocol uses Somewhat Homomorphic Encryption to generate the multiplicative triples. This is very expensive and causes the SPDZ protocol to be somewhat infeasible for larger functions. MASCOT in comparison uses oblivious transfer for the multiplication triples. The online phase of the MASCOT protocol is the same as SPDZ.

We choose to use MASCOT for the implementation of our function because of its high level of security combined with its relatively inexpensive costs in comparison to other protocols. Additionally, it must be noted that computation takes place over a finite field $\mathbf{F}_p$ for some prime p. This allows "integer-like" computation [4], which is important for our function. There was one additional protocol with similar properties as MASCOT in SPDZ-2[1]. The main difference is that SPDZ-2 does computation in modulo 2^k instead of modulo p. MASCOT was ultimately chosen as it is slightly more efficient[1].

2.2 MP-SPDZ

In order to implement the function for private meeting scheduling, we must first discuss the framework that we use to implement it. MP-SPDZ is a framework extended from the SPDZ-2 implementation to include 34 MPC protocols[4]. MP-SPDZ is a well documented GitHub repository designed to help make running and benchmarking MPC protocols easier. The main contribution and advantage of this framework as compared with previous existing MPC frameworks is the ability and ease to use different MPC protocols.

MP-SPDZ implements 34 protocols using an easy to use Python based high level interface. The framework divides its protocols by security including Malicious, covert and semi-honest, as well as into dishonest majority and honest majority settings. In all, there are MPC protocols implemented in MP-SPDZ, to suit most use cases. This allowed us, when implementing our function, to easily pick the protocol that was most appropriate and efficient for our use case.

In addition to providing many MPC protocols, MP-SPDZ allows a very easy to use python like language to create and run the function. As is required for most MPC protocols, the high level language allows users to add and multiply secret values. In addition, MP-SPDZ's implementation allows users to perform comparison and access random secret values, both of which can be important for specific functions. We will see this in greater detail in section 3 when discussing the function. As well as being able to run these operations, the language also allows for typical python like uses such as arrays and loops. The authors use a template for each of the MPC protocols implemented, so that the user can work with one single interface and access any of the appropriate MPC protocols in MP-SPDZ. This interface is built on top of an efficient low level implementation of each of the protocols. They show that the MP-SPDZ implementation beats out most of the previous competition when comparing efficiency [4]. Additionally, MP-SPDZ was the only framework which implemented so many protocols.

MP-SPDZ was chosen for the implementation of our function due to its flexibility, efficiency and trusted implementations. This flexibility, allowed the testing of multiple MPC protocols, before the ultimate selection to use MASCOT (as described in section 2.1), was chosen for the implementation. Additionally, the ability to perform comparison and select random secret values, proved to be crucial to our functions ability to run.

2.3 Networking and Communication

We use a slight modification to the traditional MPC setting that nevertheless preserves correctness and privacy [2]. In an MPC protocol, all the parties act both as clients and servers, and they all have an active role in the computation of the functionality.

In order to have a usable app, we would like the users to have a fast and user-friendly interface. The traditional MPC approach is undesirable in this respect, for several reasons. Firstly, all the users will have to participate in the preprocessing stage, which can slow things down. Secondly, all the users will need to have access to code implementing MASCOT, so there is additional overhead caused by the users installing the MP-SPDZ framework. Finally, designing distributed networked applications is significantly easier if the role of the client and server are separated.

The approach proposed in [2] separates the roles of the clients and the servers. In this paradigm, n servers act as parties in the traditional MPC setting and jointly compute the functionality. The clients, who are the voters in our setting, only send their inputs and receive their outputs. This poses obvious security problems: even though the MPC protocol is secure, the servers can modify the final output before sending it over to the clients. Securely sending the client values to the servers is also an issue.

The solution proposed in [2] relies on the ability of the preprocessing stage to generate shared randomness. More precisely, the preprocessing stage can be configured to generate shares $\langle r \rangle$ where r is a random element of the field that is unknown to all the players.

At the end of the MPC protocol, the servers jointly hold a share $\langle y \rangle$ of the output y. Summing these shares up gives y. Now, to ensure to the clients that the computed shares are not tampered with, the servers compute $\langle w \rangle = \langle yr \rangle$ and jointly send their shares $\langle y \rangle$, $\langle r \rangle$ and $\langle w \rangle$ to the clients, who can then check that $w = yr$. In this scheme, r is used as an authentication tag for y, The authors also authenticate r in the same way to defeat a subtle attack that gains information from the behaviour of the clients when the triple (w, r, y) is not well-formed.

Clients can send their values to the servers securely in a similar fashion. Using the previous methods, the servers can reveal a random value r to the client with themselves knowing only $\langle r \rangle$. The client can then broadcast their input x by broadcasting $x - r$ to the servers. Since r is random the servers don't learn anything about x, but they can nevertheless compute $(x - r) + \langle r \rangle = \langle x \rangle$, and initiate the MPC protocol from there.

3 PROTOCOL

Now that we have explored the MPC protocol, implementation framework and communication related to the function we would like to implement, it is time to discuss the function itself. As mentioned in the introduction, we would like to create a private meeting

scheduler, where clients hide and send their schedules, before jointly computing the optimal meeting time.

Before discussing the different versions of the function, as well as additions we made to the base version, we must discuss the goals of function. The goals of the function come in two parts, the inputs of the function and the outputs of the function. This will be discussed in subsection 3.1. In subsection 3.2, we will discuss the base version of the protocol and how it meets the goals of the function. After that, in subsection 3.3, we discuss the need for a check to make sure the users' inputs are valid and how we add this to the original function. Finally, in subsection 3.4, we discuss a separate function with a different idea of optimality from our original function.

3.1 Inputs and Outputs

When looking at the inputs of the function, in an ideal sense we would like to be able to represent a persons schedule in a way that can be used in the function. We chose to represent a schedule as a binary vector, where each element of the vector corresponds to a time in the schedule. For each element of the binary vector, a 0 would represent that the person is busy and cannot meet in that timeslot, whereas a 1 would represent that the person is free to meet. For example, if a person wanted to schedule a 1 hour meeting on a given day, between 9 am and 5 pm, the day would be broken down into 8 one hour chunks. Each of these 1 hour chunks would correspond to an element of a binary vector of length 8, where the person would input a 0 if they are busy during that hour and a 1 if they are available to meet. We will explore the need for this vector to be a binary vector in greater detail in subsection 3.3, and how the function solves the potential issue of a client not submitting a binary vector.

Next we must discuss the goals for the output of the function. The first and most important output of the function is the time of the meeting. This corresponds with the index of the vector which is an optimal time to meet for the users involved. Additionally, we also want the users to know how many people are able to attend this meeting. This is important, as we want the users to be informed as to whether the meeting will be a full house, or only a couple of people. this will be important for the planning of the meeting (location, resources etc.). It is worth noting that if the members schedules perfectly do not line up, that this value may be 1, which must be conveyed to the members, as they will be all alone if they attend. The final output of the function will be a confirmation that each of the users has acted honestly with their inputs to the function. It is important for the trustworthiness of the results that the users involved in the scheduling computation have acted honestly with their inputs. Therefore we will check to make sure the inputs are legitimate. The final output will be the verification of correct inputs for each user involved. This will give the users a confirmation over whether they can trust the meeting time is correct.

3.2 Base Version

The first version of the protocol we explore will be the base version. Let us first recall that the goal of the function is to compute the optimal meeting time, given the schedules of the parties involved in the meeting. First we must discuss the notion of optimality when referring to an optimal meeting time.

In the base version of the function, we consider the optimal meeting time to be the time where the most possible parties have availability in their schedule. There are other versions of optimal, which are considered below, however as a main functionality for the scheduler, we believe this to be the most commonly used when scheduling meetings in the real world. If we recall from section 3.1, the inputs of the function are a binary vector from each party, where a 0 represents the inability to meet during the given time and a 1 represents the ability to meet during that time. What this means for our function is that we would like to find the index which contains the most 1 values, across all of the binary vectors.

Before the function begins, we assume that the function receives as inputs a hidden version of the binary vectors as specified in section 3.1. The hiding procedure follows from MASCOT's protocol, where parties will create shares of their private inputs using the INPUT function, before sharing these shares with the computing parties (in our case, this will be multiple servers). Additionally, we assume that a sufficient number of multiplicative triples (and all other relevant preprocessing values) are created prior to the function running. This will allow all available arithmetic operations to be used in the function.

To do this, we divide our function into 3 parts. The first step is to add all of the binary vectors together. Given MASCOT's addition functionality, this is straightforward enough. To perform this addition step, we must first add two of the parties binary vectors together and store the value. For each additional party involved in the computation, we add their binary vector to the stored value and store this new added value instead. MP-SPDZ's implementation helps to make vector addition straightforward[4]. We will call the outcome of this first step $s = x_1 + x_2 + ... + x_n$, where x_i is party i's binary vector. It is worth noting, that this computation is performed modulo p, a large prime, so s, will (often) not be a binary vector.

The second step in our function is to find the maximum element in s and obtain both the index of this element and the maximum value. We find that since the value of s at any given index represents the amount of people who can attend the meeting at that time. This is because we restricted the users inputs to a binary vector where a 1 represents availability to attend the meeting. Therefore, by finding the maximum value, we are finding the maximum number of people who can attend the meeting. By taking the index of this maximum value, we find the timeslot where the most people are available. These are the two values we are looking for.

To find the maximum value, we begin by storing the first element of the vector s and the value of this element. We then loop through subsequent elements of the vector and compare the two values. If the stored value is less than the new value, we replace it, otherwise, we keep the stored values as is. When storing the values, we store a tuple consisting of the maximum value and the index of this maximum value. To note, in order to perform comparison, MP-SPDZ uses [8] for comparison in a prime modulus. This paper helps to allow for non-linear computation.

The output of the function so far is exactly what we desire, however there is still one more step to include. This is a subtler point that will be explored further in section 4, however it is important that there be some randomness component when choosing the meeting time. This is for security reasons and will be explored later.

In order to include the random component, we will use some randomness to break ties among the timeslots. Above it is mentioned that when looping through the vector, trying to find a maximum, that if the new element is larger than the old maximum, there is a swap, otherwise there is not. Here we add one more condition. If there is a tie between the new element and the old maximum, we would like to swap with probability $\frac{1}{2}$.

For this, we begin by generating a secret random bit. This is a function implemented in MP-SPDZ. In addition to the previous greater than check, to determine if we have a new maximum, we will add in a second check. The second check, checks to see if the new element multiplied by the secret random bit, is equal to the previous maximum. If it is, then we perform a swap, otherwise we do not. This functionality will then cause a swap half the time that the two values are equal, which is exactly what we would like to happen.

In summary, our maximum value calculator, will loop through each element in the vector. If the new element is greater than the previous element, we will swap the tuple (consisting of value and index), if the new element is equal to the previous element, then with probability $\frac{1}{2}$ we swap the tuple and finally if the new value is less than the previous element, we do nothing.

3.3 Binary Vector check

Now that we have a function that provides us with the optimal meeting time and the number of people who can attend the meeting, given that the parties submitted binary vectors, we must now perform a check, confirming that these vectors are in fact binary vectors. To see why this is so crucial, we must examine what happens if a party does not submit a binary vector. We find that given the structure of our function, a party can easily manipulate the outcome of the function and thus, forcing a meeting time to a time of their choosing.

For example, say there are 5 people involved in the meeting, thus 5 parties send forward their schedules. Now say that 4 of these parties are acting honestly and send binary vectors corresponding to their schedules. If the fifth party chooses to be dishonest, they could easily manipulate the meeting time. Say the fifth party wants to meet in the first timeslot, they can send the following vector $[10, -10, -10, ... - 10]$. Given that the rest of the parties acted honestly, no matter what the other parties selected, the first timeslot value of 10 plus the sum of the other 4 parties, must be greater than the rest of the values in the vector after summation. This is clearly problematic and shows why there must be checks and balances in place to verify to the parties that all of the other parties were honest with their submissions.

In an ideal sense, this binary vector check will inform all users of whether or not each user provided a binary vector. The user receiving these outputs can then choose to trust the output if all parties were honest by submitting a binary vector. If even one of the other parties were dishonest with their inputs, the remaining parties know not to trust the output (and maybe run the protocol again, without the dishonest client this time).

To perform the binary vector check. we run through each element of the binary vector, and check if the values are greater than or equal to 0, and less than or equal to 1. We use the same method for comparison in MP-SPDZ as is used in section 3.2 above. This binary vector check is performed at the beginning of the algorithm, and is revealed at the end alongside the rest of the output of the function.

3.4 Douglas Mode

The above protocol is built on the assumption that the optimal meeting time is based on the time where the most people invited to the meeting, can attend the meeting. In this section we propose a different definition of an optimal meeting time, as an alternative to that presented in section 3.2. The second kind of meeting being addressed here is where one member must be in attendance, we call this member Douglas. Say we have a class at school which wants to find a time to meet. The optimal meeting time for the class no longer becomes the time where most people can meet, because the time where most people can meet, may not include the professor. In this case, the optimal meeting time is when the most amount of students in the class can meet at a time that the professor (Douglas) can also meet. This idea of optimality does not only apply in a class setting, but also in a work setting where the boss must be in attendance, or even in the setting of a talk or presentation, where the presenter must be there.

While the implementation at present only explores the case where one member of the meeting must be in attendance (a singular Douglas), the same idea can be used to find an optimal meeting time where more than one member must be in attendance (multiple Douglases). In a similar way, there can also be a version of the protocol, where one of multiple people must be in attendance (one of many Douglases). However, these two variations, have not yet been implemented. We will describe briefly the additional steps required for these 2 additional protocols.

For each of these versions of the function, there is one crucial step to be added to the protocol. If we recall, the main part of the function works as follows: first we add the binary vectors together. Next, we find the maximum element in the summed vector (with some randomness in the case of a tie). To alter this protocol to fit our new use case, we add one additional step. We must multiply our sum vector s by Douglas' binary vector before looking for the maximum. What this will do is for all times that Douglas cannot meet, the summed vector s, will be set to 0 at those indices. The summed vector s, will remain unchanged at all indices where Douglas can meet. This simple multiplication step is enough to adjust the protocol to the new optimal setting.

This protocol can be extended to the case of multiple key members (Douglases) needing to attend the meeting, simply by multiplying each of these key members vectors, by s, before taking the maximum. The intuition behind this is the same. If any of the key members cannot meet at a given time, when their vector is multiplied by s, it will be set to 0. Thus, when taking the maximum, the only non-zero elements will be the elements where all key members can meet.

The final version of this protocol is when a minimum of one of potentially many Douglases must attend the meeting. This version requires a bit more work. For this version, we must add each of the many Douglases vectors together. However, one key step here is that we must cap the addition step at 1. This is possible in MP-SPDZ

implementation of MASCOT, as we can first add all of Douglases binary vectors together, and then run through each element of the new vector. Since we can perform comparison, we can then check if any element is greater than 1. If it is, we set it to 1, otherwise we keep the element the same. Now we have a new binary vector. This binary vector contains a 1 if any one of the potentially many Douglases are available to meet at that given time, and a 0 otherwise. With this new binary vector, we multiply it by s, before looking for the maximum element. This will set any time where no Douglas can meet, to 0, and thus not be included in the maximum calculation.

The first version of the Douglas protocol has been implemented. The second and third, as of now, are purely theoretical. An addition to the implementation will come.

4 SECURITY ANALYSIS

In this section we analyze the security of our protocol. In section 4.1 we recall our function and establish some basic facts about it. In section 4.2 we state some security properties that we want to establish for our protocol, discuss how many of these properties are automatically guaranteed by the MASCOT protocol, and prove what remains to be shown.

4.1 Basic facts

First we establish some terminology and basic results about our protocol. In particular, we analyze the effects of our randomized tie-breaker as discussed in section 3.2.

We denote the ith party's binary vector as x_i. Our function as defined in section 3.2 is denoted by F. The result of $F(x_1, x_2, \ldots, x_n)$ is a 3-tuple (F_1, F_2, F_3). F_1 is an index j for which $(\sum_{i=1}^{n} x_i)_j$ is maximum. F_2 is then the value of this maximum. F_3 is a list `Valid` of n elements, where $\texttt{Valid}_i$ equals 0 if party i has cheated by not inputting a binary vector, and equals 1 otherwise. Suppose that there are n ties with ordered indices $t_1, \ldots, t_n$ for the maximum. Then the protocol chooses a random index r through the following n-step process. In the first step, $r = t_1$. In the ith step, the index t_i will be assigned to r with probability $1/2$. Otherwise r is unchanged. At the end of the nth step, the randomly selected index is r.

Theorem 4.1. *Let n be the number of ties for the winning index in either base mode or Douglas mode, with the indices $t_1, \ldots t_n$. Then $P(F_1 = t_1) = \frac{1}{2^{n-1}}$ and the probability of returning any other t_i is $\frac{1}{2^{n-i+1}}$.*

Proof. For every index t_i, there are $t - i$ indices succeding it. To these correspond $t-i$ binary choices, and each choice has probability $\frac{1}{2}$ of discarding the index t_i. For the indices t_2 through t_n, there is an additional bit of uncertainty, namely if we include t_i in the first place, or toss it right away. □

We note that even though a random index is selected, it is biased towards the tail-end. In future work, we plan to fix this so that the probability of choosing a winning index is uniformly distributed.

4.2 Security definition and MASCOT security

We allow our adversary to statically corrupt up to $n - 1$ parties. These parties can arbitrarily deviate from the protocol [6]. We would like our protocol to satisfy the following security properties.

- **Privacy:** The malicious adversary cannot learn the full schedule of the interrupted party.
- **Correctness:** The result of the computation is guaranteed to be either the time-slot with the highest number of people willing to attend, or otherwise an indicator that the result is not correct.

In addition to these cryptographic security guarantees, we also need to ensure that our function corresponds game-theoretically to a voting process. That is, the optimal strategy for each player is to input 1 for the indices in which they are available to meet, and input 0 for the other indices.

To satisfy these security requirements, it is necessary to restrict our analysis to a single run of the protocol. If we allow an arbitrary number of runs, then it is easy for the adversary to break both security properties. To break privacy, the adversary can simply run the protocol many times, setting all of its inputs to 0. The adversary can then recover all of the uncorrupted parties indices with high probability, as each run of the protocol returns a randomised index for which the uncorrupted party can meet. A similar attack would work with a deterministic tie-breaker. This attack relies only on the output of the function F, and so it is inevitable.

Notice our weakened notion of correctness. The reason we cannot guarantee correctness of input in all cases is that we are working strictly within the MPC framework, where any input is allowed. In particular, as discussed in section 3.3, if the inputs are not binary vectors, then the maximum index does not correspond to the time at which the maximum number of people can meet. Our binary vector check within the MPC framework does not prevent the adversary from doing this, but at least it informs all the parties of the incorrectness of the output.

MASCOT is secure against a static malicious adversary that corrupts up to $n - 1$ parties in the real vs. ideal paradigm [5]. That is, it provides indistinguishability of the real protocol execution from the ideal case where we have access to a trusted, uncorruptable third party. This already provides the following weak notions of privacy and correctness [6].

- **Privacy:** The only information that the adversaries can learn about the party's schedule is information they obtain from the output $F(x)$.
- **Correctness:** All the participants are guaranteed to receive the output $F(x)$ as defined in section 3.2.

As MASCOT guarantees these weaker notions of correctness, it suffices to show that these notions imply the stronger requirements that we stated at the beginning of this section.

4.2.1 Proving privacy : Function leakage. In this section, we give some intuitive arguments for why the function F does not leak much about the schedule of the honest party. Our arguments fall short of being a formal proof, which we leave as interesting future work. Before giving our arguments, we note that any useful scheduling protocol should give the information that our function F provides, namely a meeting time where the most number of parties can meet, so in a sense we cannot do much better. We do try, however, to avoid some common pitfalls by randomising our tie-breaker.

The adversary controls n-1 parties. So the adversary knows the inputs $x_1, \ldots, x_{n-1}$ where the x_i's are the vectors input by the

parties controlled by the adversary. The adversary will also know the result $F(x_1, \ldots x_{n-1}, x_n)$. We claim that these do not allow the adversary to learn x_n. Let $s = \sum_{i=1}^{n-1} x_i$. Since the adversary already knows who has cheated and this sum, they already know the results F_3. Therefore, we need only consider information leaked from F_1 and F_2. Let S_2 be the maximum of the elements in S, and let S_1 be the indices in S achieving this maximum. Note that we can either have $F_2 = S_2 + 1$ or $F_2 = S_2$. We consider these two cases separately.

- $F_2 = S_2$. In this case, the adversary knows that $\{i|(x_n)_i = 1\}$ has no intersection with S_1, so they know that $(x_n)_i = 0$ for all $i \in S_1$. They also know that $(x_i)_{F_1} = 1$ and that with probability 1/2, this is the last index j for which $x_j = 1$ and $S_j = S_2 - 1$. But the adversary cannot learn anything else from this, as we are restricting to single runs of the protocol. Also, note that the probability of this outcome happening decreases as the size of S_1 increases, so while large sizes of S_1 can reveal quite a lot about the honest party's schedule, the probability of this happening is low.
- $F_2 = S_2 + 1$. in this case, the adversary knows that with probability 1/2, F_2 is the last index j for which $(x_n)_j = 1$ and $j \in S_2$. Depending on the indices in S_2 that succeed j, the adversary can guess the value of x_n for these indices. The adversary has no way of knowing what the previous indices are though, so they can't learn much.

4.2.2 Proving correctness. MASCOT guarantees that the output of $F(x)$ will be correctly received by all the participants. Notice that $F_1(x)$ is the time-slot with the highest number of people willing to attend if and only if all the parties outputted binary vectors. But F also indicates if any parties didn't output binary vectors, so our notion of correctness is easily implied by the correctness provided by the MPC protocol.

4.2.3 Other notions of security. Notice that our protocol is "game-theoretically secure", in the sense that if we restrict ourselves to one-round voting, the optimal strategy for each voter is to be truthful. From the client's point of view, the only way to make F_1 be a desired index is to input a 1 at that index, and the only way to prevent the winner from being an undesired index is to input a 0 there.

5 EFFICIENCY

The function as described in section 3 has been implemented in the MP-SPDZ framework as described in section 2.2. The ExternalIO examples were used as the backbone for our functions implementation. Efficiency can be discussed both theoretically and experimentally. We will focus on the experimental side of efficiency where we show that for most real world use cases, the function is feasible.

For the efficiency tests, we used a Dell Vostro 5402 laptop. The laptop has an 11th generation Intel(R) Core i7-1165G7 processor at 2.80GHz 1.69GHz. Additionally, there is 16.0 GB of RAM. Windows 11 Pro is installed on the machine. However, our program (the MP-SPDZ framework) must be run on either Mac or Linux. Thus, in order to run the program, a virtual machine was used. The virtual machine used was an Oracle VM VirtualBox. The virtual machine had Ubuntu 20.04 installed and 10 GB of RAM with 4 processors. The tests described were all conducted on this Virtual Machine with the specifications as described above. Table 1 is the results of these tests.

Table 1: Time (seconds) taken to run the Base version of the protocol, with the binary vector check

Vector Length	Number of Clients				
	2	5	10	15	20
6	3.4	5.8	9.9	12.9	17.5
20	3	5.9	10.1	12.1	17.3
40	3.3	5.5	10.5	13.7	16.3
80	3.9	6	9.8	13.6	16.9
160	3.3	5.9	9.9	13.4	17.1
320	3.5	5.8	9.8	13.8	18.7
640	3.5	6	9.7	13.1	17.4
1280	3.9	5.9	9.9	13.4	17.5
2560	3.7	6.2	11.2	15	19.2
5120	4.1	7.8	13.4	19.2	25.7
10240	7	13.6	26.5	38.1	41.9
20480	16.7	39.2	77.8	114.9	127.6

Table 1 explores the effect of adding more clients to the protocol as well as varying lengths of the clients binary vectors. The number of clients corresponds with the number of people involved in the meeting and the length of the binary vector corresponds with the how many potential meeting timeslots exist. The tests conducted involved 2, 5, 10, 15 and 20 clients. The binary vector length consisted of 6, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240, 20480. In the real world, we would rarely need so many potential meeting timeslots, however we wanted to run the tests until the machine started to require more time to run.

First, we notice that the changes in runtime appear to be down the columns and across the row. We will not analyze the diagonals as much as the primary relationships between values are horizontal and vertical. In other words, the effects of changing both the column and the row simultaneously are not noticed significantly.

First we explore the effects of changing the length of the binary vector, on runtime. To begin, we will fix one of the columns and examine how the runtime changes as we move down the column (thus, increasing vector length). Let us fix the first column with 2 clients involved in the scheduling. We see that runtime remains mostly constant when using vector lengths of 6 to 2560. There are some fluctuations, presumably due to the machine running the tests, however we do not see much (if any) change in runtime in this range. Increasing now to a vector length of 5120 yields the first noticeable change in runtime, albeit a small increase. This continues as we change to a vector length of 10240, which yields close to an increase of double the previous amount. Similarly, by doubling the vector length again to 20480 yields more than double the runtime of the previous length. Now, if we move across to different columns in the table (differing amounts of clients) we see a very similar trend, where the first noticeable increase occurs at a vector length of 5120, 10240 yields approximately a 1.75 times increase and 20480 yields approximately a 2.5 times increase on the values above.

This analysis on the vector length tells us that the runtime bottleneck of the function is not the associated with the length of the

vector until the length of the vector exceeds approximately 2560. This is important to note for our meeting scheduler use case of the function, since 2560 potential timeslots for a meeting would likely exceed the majority of real world meetings. Theoretically, once the runtime bottleneck becomes the vector length, we would not expect exponential growth. There are two parts of the function which are associated with the length of the vector. The first being vector addition. Given n clients, by doubling the length of the vector, we must make 2n more addition calculations. The second part of the function associated with the length of the vector is finding the maximum element of the sum of the vectors. This entails twice as many random bits being generated and twice as many comparison checks between elements. Additionally, the binary vector check will add 2n times as many comparisons for a doubling of the vector length. In summary, by doubling the length of the vector, we should expect 2n times more additions, 2n times more comparisons from the binary vector check, twice as many random bits being generated and twice as many comparison checks being generated. However, as we can see with the experimental results, this does not start to become an issue until approximately a vector length of 2560.

Next we look at the affect, on the runtime, of adding more clients to the protocol. As seen in our analysis on vector length above, it does not make a difference if we take any vector length less than 2560, however for our analysis we will begin by fixing a vector length of 6. We see that an increase from 2 to 5 clients yields an increase of approximately 2.5 seconds. After that, every increase of 5 clients yields between a 3 and 4 second increase on the runtime. This is also consistent as we increase the vector size to 2560. This shows a constant increase in runtime when increasing the clients. Even as we move to the bottom half of the table, we see that the increase in runtime is approximately constant with the increase in number of clients. The constant number, however is dependent on the length of the vector.

Let us explore what is causing the increase in runtime. As mentioned above in the vector length analysis, there are two places where an increase in clients will increase the runtime. The first being the binary vector check. For each client, a new binary vector check must be performed. As well, for each new client, there is an additional addition step being performed when adding the vectors together. These are the only places in the function itself where the number of clients affects runtime. One additional place where the number of clients affects runtime is in the preprocessing phase of the protocol. The more users involved in the protocol, the more preprocessing that has to be performed. For each new client, the relevant preprocessing for hiding the initial vector, as well as multiplication triples, have to be generated. This can be seen in greater detail in the MASCOT and SPDZ papers[3, 5].

Additionally, as mentioned in section 3.4, we have also implemented one version of the Douglas Protocol. The version implemented is the version of the protocol with a single key member who must be at the meeting (singular Douglas model). This version of the protocol includes one additional vector multiplication step. Since we are working in the singular Douglas model, we find that only one additional vector multiplication is required no matter how many clients are involved in the process. We find that the length of the binary vectors (within our testing boundaries), plays no noticeable role in increasing the runtime of this version of the protocol, in comparison with the base version. Overall, we find that the runtime does not increase in any meaningful way with the Douglas Protocol in comparison with the Base Protocol.

In summary, we see that while both the vector length and the number of clients affect the runtime of the protocol. In most practical settings, the runtime is not affected by the vector length as it does not bottleneck until vectors of length 2560 or greater. However, we should note that an increase in the number of clients involved in the meeting will lead to increases in the runtime. While the increases are only constant, this can add up very quickly with large meetings. For example in a large townhall type meeting where a company may invite 100 people, the computation will take over a minute to run. This may still be a very feasible amount of time and even expected given the privacy that comes with this function, however it is worth noting that it is not instantaneous like most things on the internet in the current year.

6 CONCLUSION

The MASCOT Private Meeting Scheduler provides a solution to the problem of being able to schedule a meeting (or appointment) without the need to send forward the members' schedule. The meeting time is determined with members only sending shares of their hidden schedule (in the form of a binary vector) to computing parties. In the end, members will receive the time of the meeting, the number of people able to attend the meeting and a validity check for each other member of the meeting, confirming that their inputs are valid. If the inputs of even one other member are not valid, or in other words, if they do not send a binary vector, this will appear in the output and will tell all other members not to trust the results.

The private meeting scheduler utilizes MPC and specifically, the MASCOT protocol, to perform this private computation [5]. MASCOT provides malicious security against a dishonest majority. This is the strongest available security. The protocol implementation leverages the MP-SPDZ framework [4] for MPC, as an implementation of MASCOT and more generally, the overall structure.

When using an MPC protocol such as MASCOT, we find that a number of the security properties are guaranteed. In this paper we explore the areas of the protocol which can still be subject to attack. In the security analysis, a security definition is provided, giving a privacy and correctness guarantee. We provide information on the function leakage and conclude the security analysis with a sketch of the game theoretic proof of security. We find that there is a certain amount of data leakage, such as the availability at one single timeslot can easily be retrieved from the computation, however we find this to be both unavoidable and reasonable, since a client at the end of the day wants to find a timeslot that works for all parties involved.

As mentioned, the protocol was implemented with the help of the MP-SPDZ framework. The efficiency of the algorithm was explored in an experimental way, based on the implementation. We find that the key major factors affecting the speed of the protocol run, the length of the binary vector (number of potential meeting times) and the number of members involved in the meeting. Our findings suggest that the number of clients affects runtime in a linear manner. We also find that the length of the vector plays no

role in the runtime until a length of approximately 2560. We claim that for our use of the function, this is immaterial as meetings very rarely have over 2560 potential times. In addition to providing an experimental view on efficiency, we also aim to provide some reasoning as to why our results make sense, based on our function.

We believe this function and implementation to be feasible to run and secure and correct for clients wanting to schedule a meeting.

6.1 Future Work

Future work for this function can be explored from multiple angles. First, we can look at implementing this function into a working application. Second, we can work on implementing different versions of the function.

The first area of potential future work is turning this working protocol and demo into a user application. The idea here would be to have a user interface that clients can open up and select their meeting times. The interface would then hide the schedule and send shares to a set of computing parties. Once the computing parties have received enough of these shares, the computing parties would run the computation, and the results would be outputted to the client. This is very similar to what has already been implemented, only without the proper networking and communication, as well as user interfacing.

The second piece of future work has been addressed briefly when discussing the protocol itself. Our protocols work under two very specific definitions of an optimal meeting time. While these are likely to be the most commonly used in practice, there are other definitions for the best time to meet. There were two listed earlier in the paper, being with multiple key members who must attend the meeting and where there are multiple key members but only one of them must attend the meeting. We believe that these two versions of the protocol can be important depending on the use case. Additionally, we believe that these are not the only definitions of optimal for meeting times, and future work is to identify and implement these definitions as well.

In all, we believe that there are many future prospects for this work and the work thus far is only the beginning.

REFERENCES

[1] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SpdZ$_{2^k}$: Efficient mpc mod 2^k for dishonest majority. *Lecture Notes in Computer Science*, page 769–798, 2018. doi:10.1007/978-3-319-96881-0_26.

[2] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. Cryptology ePrint Archive, Paper 2015/1006, 2015. https://eprint.iacr.org/2015/1006. URL: https://eprint.iacr.org/2015/1006.

[3] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. *Lecture Notes in Computer Science*, page 643–662, 2012. doi:10.1007/978-3-642-32009-5_38.

[4] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020. doi:10.1145/3372297.3417872.

[5] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016. doi:10.1145/2976749.2978357.

[6] Yehuda Lindell. Secure multiparty computation (mpc). Cryptology ePrint Archive, Paper 2020/300, 2020. https://eprint.iacr.org/2020/300. URL: https://eprint.iacr.org/2020/300, doi:10.1145/3387108.

[7] Christian Mouchet, Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, and J Hubaux. Lattigo: A multiparty homomorphic encryption library in go. In *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2020.

[8] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. *Public Key Cryptography – PKC 2007*, page 343–360. doi:10.1007/978-3-540-71677-8_23.

[9] Parsa Salimi and Jonathan Gold. Mascot Private meeting scheduler, 8 2022. URL: https://github.com/parsa-salimi/MP-SPDZ-Max.

[10] M.C. Silaghi. Meeting scheduling guaranteeing n/2-privacy and resistant to statistical analysis (applicable to any discsp). In *IEEE/WIC/ACM International Conference on Web Intelligence (WI'04)*, pages 711–715, 2004. doi:10.1109/WI.2004.10147.