# Towards Secure Evaluation of Online Functionalities (Corrected and Extended Version)

Andreas Klinger
RWTH Aachen University
Aachen, Germany
klinger@itsec.rwth-aachen.de

Ulrike Meyer
RWTH Aachen University
Aachen, Germany
meyer@itsec.rwth-aachen.de

## ABSTRACT

To date, ideal functionalities securely realized with secure multi-party computation (SMPC) mainly considers functions of the private inputs of a fixed number of a priori known parties. In this paper, we generalize these definitions such that protocols implementing online algorithms in a distributed fashion can be proven to be privacy-preserving. Online algorithms compute online functionalities that allow parties to arrive and leave over time, to provide multiple inputs and to obtain multiple outputs. In particular, the set of parties participating changes over time, i. e., at different points in time different sets of parties evaluate a function over their private inputs. To this end, we propose the notion of an online trusted third party that allows to prove the security of SMPC protocols implementing online functionalities or online algorithms, respectively. We show that any online functionality can be implemented perfectly secure in the presence of a semi-honest adversary, if strictly less than $1/2$ of the parties participating are corrupted. We show that the same result holds in the presence of a malicious adversary if it corrupts strictly less than $1/3$ of the parties and always allows the corrupted parties to arrive and provide input.

Note, this is the corrected and extended version of the work presented in [24].

## CCS CONCEPTS

• **Security and privacy** → **Formal methods and theory of security**; **Privacy-preserving protocols**.

## KEYWORDS

SMPC, dynamic, online algorithm, trusted third party, privacy

## 1 INTRODUCTION

Secure multi-party computation (SMPC) allows parties to evaluate a function over their private inputs in a distributed fashion such that each party only learns its prescribed output and anything it can deduce from combining its prescribed output with its own private input. An SMPC protocol describes the individual communication and computation steps each party follows during the distributed evaluation of the function it implements. Informally speaking, such a protocol is said to securely evaluate the function if it correctly computes the prescribed output of the function and if during the evaluation, the parties do not learn anything that goes beyond what they would learn if the function was evaluated centrally by a trusted third party (TTP) - even in the presence of an adversary.

Secure function evaluation (SFE), a special case of SMPC, considers functions that could - if they were evaluated by a TTP - be evaluated by a single run of an (offline) algorithm that takes the inputs of a fixed number of parties and computes the desired output for each party. SFE can also be extended to *reactive SMPC* to cover reactive algorithms [19] which allow an a priori known fixed number of parties to provide input over multiple rounds and obtain output in each round. In addition, the output can depend on a state, which itself depends on all previous inputs and outputs.

However, the prominent set of functions or problems that can be solved with the help of online algorithms [17] have not yet been considered in SMPC: Online algorithms receive events one after another and for each event they have to decide immediately how to deal with it. An example is the online matching problem with general arrival, where in each event a party with their preferences arrives. The online algorithm then has to decide how to match the party, e. g., match the party now or save it for later. The overall goal is typically to maximize the reward while minimizing the cost, e. g., achieve a high matching quality with short waiting times. The online algorithm has no knowledge regarding future events, i. e., the parties arriving are not known in advance, and previous decisions can affect future decisions. Therefore, the online algorithm has to store information in a state, e. g., all unmatched parties already revealed. The main difference between reactive and online algorithms is that whereas the participating parties are fixed for reactive algorithms, parties can dynamically arrive and leave in online algorithms. In addition, reactive algorithms assume that in each round all participating parties provide input and obtain output. However, online algorithms also allow that only subsets of currently participating parties provide input and obtain output, i. e., only a subset of the parties is aware that an event happened.

SMPC protocols for offline matching is an important research area [1, 6, 21, 25, 28]. However, online matching algorithms have the advantage of not requiring to restart the complete computation as soon as a new party arrives and thus the benefit of reducing waiting times. Such online matching problems arise for example when open job positions are to be filled by applicants, or students apply for internship offers and need to be told immediately whether or not they are accepted, or for providing possible dating recommendations, or finding other people for ride-sharing on the fly. All these examples require private information, which should be protected.

If parties want to securely compute an online matching algorithm over their private inputs, then they have to securely evaluate the online algorithm: When a new party arrives and provides their input, the online algorithm checks immediately whether the new party can be matched with any previously unmatched party. If there is no match, then the party and their input is stored in the state. If the party is matched, then the matched parties leave the computation and are removed from the state. We want to stress that the participating parties are not known in advance and can change over time. In addition, parties may provide input multiple times.

If we want to model the online matching example with standard techniques, then we have two main possibilities: The first is to use reactive SMPC and assume that all parties participate at all times, and some just don't provide input. The second is to use a special set of compute parties [2] separated from the parties that want to be matched. The compute parties will receive the inputs of the parties that want to be matched, perform the computation and distribute the output, all done in a privacy preserving manner. Both techniques allow to securely evaluate the desired functionality, and allow to store a state containing the parties and their inputs in a distributed fashion, either between all parties in the first case or only between the compute parties in the second case. However, the first technique is impractical as all parties are always participating and need to be present. The second technique requires an unsatisfying assumption, namely that there are a few compute parties and all parties have to trust at least a subset of them. This strong assumption seems hard to achieve in the online settings, as the set of trusted compute parties needs to be determined a priori for an a priori unknown set of participating parties. The more parties participate, the more likely it is that at least some do not trust the compute parties.

So in this paper, we want to answer the question: Can we do better, i. e., let only those participate in the computation that are actually relevant in the current evaluation, but yet hide as much information as possible? Ideally evaluating an online functionality such as online matching in a privacy-preserving way does not only entail protecting the inputs and outputs of all parties involved, and securely keeping a state over a changing set of parties, but also hiding the arrival and departure of parties from the other participating parties. This includes hiding the point in time when a party provides input and whether a party participates at all.

We therefore propose a new model for online TTPs (OTTPs) that can evaluate online functionalities, i. e., the mapping of inputs and outputs of online algorithms. In particular, we first introduce an ideal OTTP where even the arrival of the parties is private. Then, we relax the requirement, and broadcast the party that arrives and provides input (referred to as OTTP with broadcast of input parties (OTTPI)). We restrict the OTTP and OTTPI model to parties providing their input one after another.[1]

A protocol's security is expressed in terms of its ability to simulate the evaluation by one of the OTTPs. We show that for any online functionality there exists a perfectly secure protocol in the OTTPI setting if there are always at least three parties present and assuming an honest majority of parties present. The result holds even in the presence of an arrival respecting malicious adversary, if there are always at least four parties present and assuming strictly less than 1/3 of the parties present are corrupted.[2] We also show that the OTTPI setting is imperfect in the sense that public arrival can leak partial inputs which shows that the ideal OTTP setting is stronger than the OTTPI setting.[3] In Appendix B we discuss why the arrival can in general not easily be hidden in online settings, i. e., why a secure protocol in the ideal OTTP model is hard to achieve.

The rest of the paper is organized as follows: We discuss previous results w. r. t. offline and reactive TTPs in Section 2. We cover the notations used throughout the paper and preliminary definitions in Section 3. In Section 4 we model the OTTP and OTTPI, and define security. Our feasibility results are presented in Section 5. We then show in Section 6 that the ideal OTTP is more powerful than the OTTPI. The paper concludes with a summary and an overview of potential future work. We discuss the setting with a dishonest majority, and why the arrival time can in general not easily be hidden in Appendices A and B. Finally, we extend the setting to allow the evaluation of more general online functionalities, introduce the OTTPE and OTTPP, and shortly present the corresponding feasibility results in Appendix C.

Note, this is the corrected and extended version of the work presented in [24]. Major corrections and extensions are marked and explained throughout the paper with footnotes.[4]

## 2 RELATED WORK

SFE and SMPC in general has first been studied by Yao in [29] in the two party setting, and by Goldreich et al. in [20] for the multi-party setting. They showed how to securely evaluate any function based on garbled circuits and one-way functions, respectively. Shortly afterwards, other constructions have been proposed, e. g., Chaum et al. [11] simulated boolean operations and used zero-knowledge proofs for error correction, Ben-Or et al. [5] used Shamir's secret sharing scheme [27] to simulate arithmetic circuits, or Rabin et al. [26] used a verifiable secret sharing scheme and broadcast channels. Loosely speaking, the security of these approaches is based on showing that a party cannot learn anything new from participating in the protocol, compared to what it could learn from its input and output only. Beaver [4] was the first to use an explicit trusted third party to define security, i. e., he defined a protocol to be secure if it doesn't reveal more than what is revealed by an incorruptible party that receives all inputs, computes the function and then distributes the outputs. In addition, he showed that protocols can be combined sequentially. This was generalized by Canetti [8, 9] in his universal composability (UC) framework, which allows composing complex functionalities essentially arbitrarily from sub-protocols. If the sub-protocols are secure in the UC framework, then so is the composed protocol. General SMPC based on threshold homomorphic encryption was introduced by Cramer et al. in [13]. Universal composability for SMPC based on threshold homomorphic encryption was proven by Damgard et al. in [14]. Reactive SMPC [19] allows parties to execute a protocol over multiple rounds, i. e., in each round all parties provide input and obtain a

[1]In Appendix C we will remove this restriction to allow the evaluation of any online algorithm modeled as OTTP with event broadcast (OTTPE) and OTTP with public participation (OTTPP).

[2]We show the same feasibility results for the OTTPE and OTTPP setting in Appendix C.

[3]The OTTPI setting is a special case of the OTTPE and OTTPP setting, and thus the OTTPE and OTTPP setting are similar imperfect .

[4]In the original version [24] we modeled the arrival of parties too abstract, and thus with this corrected version we provide clarification, and model the arrival (and input providing) of parties more explicitly. The main change is that "receiving events" (which consists of a party with its input) is now split into an "arrival" and a separate "input providing" phase. We also corrected some other minor oversights, e. g., definition of broadcast, definition of perfect emulation, use of empty output $\perp$, definition of privacy with public arrival, and incorrect examples. Furthermore, we included additional explanations and details throughout this version.

corresponding output (possibly depending on a state). All of these results rely on the fact that the protocols are executed by a fixed and a priori known set of parties. However, online functionalities require that the set of parties can change over time, i. e., over time different sets of parties execute the protocol with different inputs. This includes that there is no fixed set of parties, nor are the parties participating known in advance. Such an "online" behavior cannot be modeled in the previous definitions. In contrast to its previous versions, the most recent version of Canetti's UC framework [10] allows adding parties or protocols dynamically, e. g., as required in distributed peer-to-peer system. The UC framework [10] could be used to model and prove the security of (certain) protocols implementing online functionalities. However, it does not allow to model our ideal OTTP that hides even the participation of any party.

Another model used to implement SMPC protocols, especially w. r. t. databases, is to use three sets of parties (not necessarily disjoint) [2]: One set of parties provide inputs and one set of parties obtain outputs, e. g., based on queries. The third set of parties is a fixed and always present set of compute parties that receives the query, performs the actual computation and distributes the output. A state is stored in a database in a secure fashion. E. g., Sharemind [7] uses three compute parties that perform the computations and additively share the database entries. Even though defined with a fixed set of parties providing input and obtaining outputs, it is rather straightforward to implement a TTP in such a model, including one for online algorithms. However, these models require that the compute parties are trusted, e. g., for Sharemind two of the three compute parties need to be trusted. This shifts the trust from a single TTP to a set of trusted peers. W. r. t. online functionalities, finding such a set of parties that will be trusted by an a prior unknown set of parties seems hard to achieve in general, as the more parties participate, the more likely it is that at least some do not trust the compute parties. Thus, we want to avoid that assumption. In addition, it does not allow to model our ideal OTTP.

Baron et al. [3] proposed a dynamic proactive secret sharing scheme, i. e., it is secure against mobile adversaries and parties can join and leave. They also mention that their scheme could be used to construct an SMPC protocol with a changing set of parties. However, they did not provide an actual construction, and it is limited to providing input only once at the beginning. Choudhuri et al. [12] recently introduced the notion of fluid SMPC, where parties can dynamically join and leave the computation. Their model consists of input, output and compute parties. However, only the compute parties can dynamically change during the computation, and no additional inputs can be provided during the computation. Thus, both schemes are effectively a realization of SFE as input and output is provided only once. Eldefrawy et al. [16] introduced basic building blocks for proactive secure SMPC protocls based on proactive secret sharing, i. e., share and reconstruct a secret, refresh it (create new shares for the same secret), recover it (re-generate a lost/deleted share), and add and multiply shares. Additionally, they show how to redistribute a share to a new set of parties, i. e., parties can join and leave. However, they assume the new set of parties is specified by a TTP (as they consider a cloud server setting). In addition, they do not provide a construction or model how to actually implement an online functionality, nor do they allow to model our ideal OTTP.

It is interesting to note that to the best of our knowledge all implementations of secure protocols for offline or reactive functionalities assume that the participating parties know which other parties participate, even though an ideal TTP could easily hide this information.[5] As parties arrive and leave over time in the online setting, it is also desirable to hide the arrival and departure of parties from the other participating parties. This seems generally hard to achieve as we will discuss in Appendix B.

## 3 PRELIMINARIES

We first introduce some basic notations and preliminary definitions.

### 3.1 Notation

We define the set of natural numbers with and without zero as $\mathbb{N}_0 := \{0, 1, 2, \dots\}$ and $\mathbb{N} := \{1, 2, \dots\}$, respectively. The cardinality of a set $B$ is denoted by $|B|$, and the power set of $B$ is denoted as $2^B$. We will use $\lceil\cdot\rceil$ to denote the ceiling function. We will use $\perp$ as special symbol to indicate no input or no output, and therefore assume for every input space $\mathcal{X}$ and every output space $\mathcal{Y}$ that $\perp$ is not included, i. e., $\perp \notin \mathcal{X}$ and $\perp \notin \mathcal{Y}$.[6]

### 3.2 Perfect Indistinguishability

Let $X = \{X(z, k)\}_{z \in \{0,1\}^*, k \in \mathbb{N}}$ be a probability ensemble, i. e., an infinite sequence of probability distributions indexed by $z \in \{0, 1\}^*$ and $k \in \mathbb{N}$ [10].

Definition 1 (Perfect Indistinguishability, based on [10]). *Two probability ensembles $X$ and $Y$ are* perfectly indistinguishable, *denoted $X \overset{p}{\equiv} Y$, if for all $z \in \{0, 1\}^*$ and $k \in \mathbb{N}$ the probability distributions $X(z, k)$ and $Y(z, k)$ are identical.*

### 3.3 Shamir's Secret Sharing Scheme

Shamir's secret sharing scheme [27] is a $(t, n)$ threshold secret sharing scheme that shares a secret $s$ among $n$ parties such that any $t$ parties can reconstruct the secret. A secret $s$ is shared among $n$ parties $p_1, \dots, p_n$ with threshold $t$ as follows: Let $q$ be a prime such that $s < q$. Select coefficients $a_1, \dots, a_{t-1} \in \{1, \dots, q\}$ uniformly at random, and construct the polynomial $g(x) := s + \sum_{i=1}^{t-1} a_i x^i \bmod q$. Party $p_i$ obtains as share $g(i)$ for $i \in \{1, \dots, n\}$. Reconstruction can easily be done with any $t$ shares, e. g., with Lagrange interpolation.

### 3.4 Online Algorithms

Online algorithms try to solve online problems where *events* occur one after another [17]. The *decision* how to deal with an event has to be made immediately on occurrence, and typically without any knowledge about future events. The goal is to minimize costs or

[5]Hiding the number of parties can increase privacy, e. g., assume the functionality defines as output the sum of all inputs. If only two parties participate, the input of the other party is immediately learned. However, if the number of parties is unknown, the input is not learned.

[6]Correction: In the original version [24] we defined $\perp$ to be part of every input and output space. However, this led to some confusion, as we used $\perp$ implicitly in two different meanings. We resolve this here by excluding $\perp$ from the input and output space, and use it only as a special symbol to indicate no input or no output. If a functionality requires "empty input/output" as actual input or output, then this can be encoded as special strings in the input and output space itself, e. g., "empty input" or "empty output". Note, this change is only relevant for the extensions discussed in Appendix C where parties can provide input multiple times.

maximize the reward for a given sequence of events. In addition, online algorithms require to support that the set of participating parties can change over time, and that it is unknown in advance. In contrast, offline algorithms are executed only once, i. e., all the information required is present from the very beginning, and only one final decision has to be made. Reactive algorithms are somewhere between offline and online algorithms, as they allow multiple inputs, but assume that the participating parties do not change.

An example for an online problem is *online matching with general arrival* [18], which we will use as a running example throughout the paper. Parties arrive one after another, and each party has a private input. Whenever a party arrives, the online algorithm has to decide which parties to match, e. g., based on some function over the inputs. Unmatched parties are saved in a state together with their inputs. If a party is matched, it is removed from the state or if it just newly arrived, never added to the state. Typically, the goal is to maximize the overall matching quality, while minimizing the costs, e. g., the waiting time of all parties.

Note, the set of parties that actually participate or the order in which they arrive is assumed to be unknown in advance. There is also no prior knowledge w. r. t. their inputs.

## 4 MODEL

Next, we will introduce online functionalities, and define the corresponding ideal OTTP and OTTPI. Then, we present our security definition and adversary model.

### 4.1 Online Functionalities

Assume a number of parties want to evaluate a function over their private inputs in a *secure* fashion. Informally speaking, secure means that each party shall only learn its prescribed output and anything it can deduce from combining its prescribed output with its own private input, and that the output they receive is correct [19]. This has to hold even in the presence of an *adversary* controlling a subset of the parties which we will refer to as *corrupted* parties. An adversary tries to deduce more information, e. g., private inputs of other parties, or alter the outputs [19].

In an *ideal world*, a function can be securely evaluated with a *trusted third party (TTP)* [19], i. e., a special entity that cannot be corrupted. The TTP will receive the private inputs of the parties, evaluate the function correctly and distribute the prescribed outputs. A TTP hides as much information as possible from the participating parties, e. g., inputs and participation. However, in the *real world* such a TTP may not exist. Therefore, the parties execute an SMPC protocol that simulates the TTP [19]. Such a protocol evaluates the desired functionality in a distributed fashion, i. e., the parties themselves perform the computation and exchange messages.

In *secure function evaluation (SFE)* [19], a fixed and a priori known set of parties $\mathcal{P} := \{p_1, \ldots, p_n\}$ want to evaluate an *offline functionality* $f : \mathcal{X}^n \to \mathcal{Y}^n$ where $\mathcal{X}$ is an input space and $\mathcal{Y}$ an output space. Given the inputs $x_1, \ldots, x_n$ of the corresponding parties, the offline functionality $f$ is evaluated, i. e., $(y_1, \ldots, y_n) := f(x_1, \ldots, x_n)$, and party $p_\ell$ obtains output $y_\ell$ for $\ell \in \{1, \ldots, n\}$. Another set of functions considered are *reactive functionalities* [19], which iteratively compute a *reactive function* $\overline{f} : \Phi \times \mathcal{X}^n \to \Phi \times \mathcal{Y}^n$, i. e., $(\phi_i, y_{i,1}, \ldots, y_{i,n}) := \overline{f}(\phi_{i-1}, x_{i,1}, \ldots, x_{i,n})$ for $i \in \mathbb{N}$ where $\phi_0 \in \Phi$ is the *initial state* and $\Phi$ is a state space. The reactive function $\overline{f}$ is evaluated in multiple rounds by a fixed and a priori known set of parties $\mathcal{P} := \{p_1, \ldots, p_n\}$. In each round all parties provide input and obtain output. The output may depend on a state that itself depends on all previous inputs and outputs. Complete definitions of offline and reactive functionalities are given in Appendices D.1 and D.2.

Here, we newly introduce *online functionalities* that define the output of an online algorithm given a sequence of events: The core component is the *decision functionality* which computes a decision and updates the state, based on an event and the current state. An event will consist of a party and their input. A decision will consist of a set of parties and their corresponding outputs.

Definition 2 (Online Functionality). *Let $\mathcal{E}$ be an event space, let $\mathcal{D}$ be a decision space, and let $\Phi$ be a state space. An* online functionality *maps an ordered sequence of events $\sigma_1, \sigma_2, \ldots \in \mathcal{E}$ to an ordered sequence of decisions $\gamma_1, \gamma_2, \ldots \in \mathcal{D}$ by iteratively computing a* decision function $\overset{\infty}{f} : \Phi \times \mathcal{E} \to \Phi \times \mathcal{D}$ *with* initial state $\phi_0 \in \Phi$. *The decision function $\overset{\infty}{f}$ is defined as $(\phi_i, \gamma_i) := \overset{\infty}{f}(\phi_{i-1}, \sigma_i)$.*

The online functionality is an abstract representation of an online algorithm, i. e., the logic without any implementation choices that do not influence the outcome. An example is to greedily match two parties with the same input based on the order they arrived, i. e., it is specified that one has to "greedily" match the parties, and parties are matched based on the order in which they arrive.[7]

Note, that the fact that an online functionality is an abstract representation of an online algorithm implies the following important difference between offline and online functionalities: For offline problems typically an optimal solution exists, but for online problems typically not. E. g., consider maximum matching, i. e., maximize the number of matches. If all parties are already present (offline setting), then one can compute all optimal (maximal) solutions and select one uniformly at random. Hence, the offline functionality can be defined as something like "select uniformly at random one of the optimal solutions". It is not important how the optimal solutions are obtained. However, in the online setting, an optimal solution for the underlying online problem is typically not achievable.[8] Therefore, in general it is not useful to define an online functionality as "selecting the optimal solution", as one could never specify a protocol computing that solution at all. Defining an online functionality as only "selecting an approximate solution" is not useful either. Any online algorithm has to decide immediately for each party how to match it, i. e., it has to follow a certain (matching) strategy, e. g., match greedily. Consider maximum matching with a matching strategy where newly arriving parties are just greedily matched if possible, and otherwise they are kept for later. The problem is that there are multiple strategies to match a party newly arriving with one of the parties which are already present, e. g., by selecting a match based on the order they arrived, or alternatively in reverse order. No matter which strategy is used, in the end it is just one strategy, which can be different from what is

[7]Correction: In the original version [24] the provided example was wrong, as it actually influenced the outcome.
[8]In general, an optimal solution can only be computed after all parties have arrived. However, then it is no longer an online problem.

defined in the online functionality. Hence, the online functionality cannot be realized if it is not defined precisely enough. Therefore, any online functionality solving an online problem has to specify such a strategy, too, i. e., it has to define the online algorithm that computes a certain solution.[9]

Another main difference between online and offline functionalities is that offline functionalities represent computations where a fixed set of parties meet once, provide input once and receive output once. Online functionalities are similar to reactive functionalities in that they allow providing multiple inputs and receiving multiple outputs dependent on some state that can be updated. However, parties do not have to participate from the very beginning, but can arrive at a later point in time. A party can also leave, e. g., when it is matched, while other parties will continue the evaluation of the online functionality. Thus, the set of parties participating can change over time. In addition, in each round a reactive functionality requires input of each party and delivers output to each party. This implies that all parties know exactly when a new round starts. In contrast, online functionalities also allow that only a subset of the participating parties provide input and a possibly different subset of participating parties obtains output, i. e., it is not necessarily known when and if an event happens.

If parties want to evaluate an online functionality, then they provide their inputs one after another. After each input they compute the decision, distribute the outputs and update the state. An event $\sigma_i$ represents a party $p_i^{\text{IN}}$ and their corresponding input $x_i$. A decision $\gamma_i$ represent the set of parties that receive outputs $\mathcal{P}_i^{\text{OUT}}$ and their corresponding outputs. An ordered sequence of parties arriving and providing inputs defines an ordered sequence of events $\sigma_1, \sigma_2, \ldots \in \mathcal{E}$. As parties *arrive* one after another, they do not participate in the evaluation from the very beginning. A party can also *leave* and thus discontinue the evaluation.[10] After a party has left, it will no longer receive any output. How and when a party leaves is defined by the online functionality, e. g., after the party is matched or based on some special input. A party that has arrived and not left after a decision is *present* and needs to be stored in the state. Any party that arrives or is already present *participates* in future computations until it leaves. If a party leaves, then it is no longer present, nor will it participate in future computations. Note, we restrict the parties here to provide input exactly once. We will extend that later in Appendix C. As the decision depends only on the event and the current state, but any participating party can receive output, the state has to store the set of parties present.

Definition 3 (Arrival of Parties). *Let $\mathcal{P}$ be a countable set of parties, let $\mathcal{E}$ be an event space and let $\Phi$ be a state space. We define $\mathcal{P}_i \subseteq \mathcal{P}$ as the set of parties* present *in $\phi_i \in \Phi$ for $i \in \mathbb{N}_0$, i. e., the set of all parties stored in state $\phi_i$ after a decision is computed. A party $p_i^{IN} \in \mathcal{P}$* arrives *in event $\sigma_i \in \mathcal{E}$ if party $p_i^{IN}$ occurs in $\sigma_i$ and party $p_i^{IN}$ is not already present in $\phi_{i-1}$ ($p_i^{IN} \notin \mathcal{P}_{i-1}$). A party $p \in \mathcal{P}$* participates *during event $\sigma_i \in \mathcal{E}$ if party $p$ arrives in $\sigma_i$ or party $p$ is already present in $\phi_{i-1}$ ($p \in \mathcal{P}_{i-1}$). A party $p \in \mathcal{P}$* leaves *in event $\sigma_i \in \mathcal{E}$ if party $p$ participates in $\sigma_i$ and party $p$ is not present in $\phi_i$ ($p \notin \mathcal{P}_i$).*

Note that we use $p_1, p_2, \ldots$ as symbols for parties, and a party is identified by $p_j$ (e. g., an ID or IP address) and not its index $j$. The index $j$ will be used to indicate relations between parties and their inputs/outputs. In addition, the index $j$ does not have to correlate to the time when a party provides input. For an event $\sigma_i$ and a decision $\gamma_i$ we use $p_i^{\text{IN}}$ and $p_{i,\ell}^{\text{OUT}}$ as symbols for parties. The first index $i$ indicates the event, and the second index $\ell$ is used for enumeration. As a party can provide input and obtain output at different times it may, e. g., hold that $p_3^{\text{IN}} = p_{5,2}^{\text{OUT}} = p_7$.

The evaluation of an online functionality with the private inputs of parties is defined as follows:

Definition 4 (Evaluation of an Online Functionality[11]). *Let $\mathcal{X}$ be an input space and let $\mathcal{Y}$ be an output space. Let $\mathcal{P} := \{p_1, p_2, \ldots\}$ be a countable set of parties. Let $\mathcal{E} := \mathcal{P} \times \mathcal{X}$ be an event space. Let the decision space be $\mathcal{D} := \{\{(p_{j_1}^{OUT}, y_{j_1}), \ldots, (p_{j_\ell}^{OUT}, y_{j_\ell})\} \in 2^{\mathcal{P} \times \mathcal{Y}} \mid \ell \in \mathbb{N}_0$ and all $p_{j_1}^{OUT}, \ldots, p_{j_\ell}^{OUT}$ are pairwise different $\}$. We define $\mathcal{P}_i^{OUT} := \{p_{i,1}^{OUT}, \ldots, p_{i,m_i}^{OUT}\}$ as the set of parties obtaining output at decision $\gamma_i$. Let $\Phi$ be a state space and $\phi_0 \in \Phi$ be an initial state. Let $\sigma_1, \sigma_2, \ldots \in \mathcal{E}$ be an ordered sequence of events, where each party in $\mathcal{P}$ occurs at most once.*

*For each event $\sigma_i := (p_i^{IN}, x_i)$, where $x_i$ is the private input of party $p_i^{IN} \in \mathcal{P}$, the decision function $\overset{\infty}{f} : \Phi \times \mathcal{E} \to \Phi \times \mathcal{D}$ is* evaluated *as $(\phi_i, \gamma_i) := \overset{\infty}{f}(\phi_{i-1}, \sigma_i)$. For each decision $\gamma_i := \{(p_{i,1}^{OUT}, y_{i,1}), \ldots, (p_{i,m_i}^{OUT}, y_{i,m_i})\}$ party $p_{i,k}^{OUT} \in \mathcal{P}_i^{OUT}$ obtains output $y_{i,k}$ with $k \in \{1, \ldots, m_i\}$ and $m_i \in \mathbb{N}_0$.*

The set of all parties $\mathcal{P}$ consists of all parties that could eventually participate, e. g., all students from a certain university or banks from a country. Note that this is different from reactive functionalities: In reactive functionalities all parties in $\mathcal{P}$ participate in the evaluation, e. g., we know exactly which students participate. Online functionalities also allow that only a subset $\mathcal{P}' \subset \mathcal{P}$ participates, i. e., we only know that students can participate, but we do not know which students actually will participate.

Note that Definition 4 explicitly allows that not all parties obtain an output and also no party at all, e. g., if $\gamma_i = \emptyset$, then $\mathcal{P}_i^{\text{OUT}} := \emptyset$ and nobody receives any output. In addition, the party in event $\sigma_i$ can but does not have to be part of the corresponding decision $\gamma_i$.

## 4.2 Online Trusted Third Party

A *trusted third party (TTP)* is an ideal entity that cannot be corrupted and behaves exactly as specified by the (offline, reactive or online) functionality to be evaluated. Given the input of parties, it returns the output only to those specified by the functionality. Ideally, it hides as much information as possible.

An *offline TTP* receives the inputs of the parties, evaluates the offline functionality, and distributes to each participating party its prescribed output. After this process, the evaluation is finished. In

[9]Extension: We added this paragraph to describe why an online functionality needs to be an abstract representation of an online algorithm.

[10]Note that in reactive evaluations the parties are always present and always participate in the evaluation. If a set of computation parties is assumed, then at least this set is always present.

[11]Correction: In the original version [24] a party $p_{i,k}^{\text{OUT}} \in \mathcal{P}_i^{\text{OUT}}$ occurring in the decision $\gamma_i$ obtained output if and only if $y_{i,k} \neq \bot$. However, as here now $\bot \notin \mathcal{X}$ and $\bot \notin \mathcal{Y}$, this condition is now removed. Note, that here this condition was actually never required, as a party not included in the decision did already never obtain any output.

contrast, a *reactive TTP* evaluates a reactive functionality by receiving inputs and providing outputs in multiple rounds, while storing the state. In classical SMPC the concepts of offline and reactive TTPs have already been introduced. However, in Appendices D.3 and D.4 we redefine them to be easier comparable with our definition of an *online TTP (OTTP)* later in Definition 5.

Note, in classical SMPC it is assumed that the participating parties know which other parties participate, even though an ideal TTP could hide them (as it is ideal). We use this convention for offline and reactive TTPs, but we break with this convention for our novel ideal OTTP, i. e., no party knows which or when other parties participate or provide input. We therefore refer to such an OTTP explicitly as *ideal* OTTP, and we use OTTP to refer to *online* TTPs in general. An *OTTP* continuously receives inputs and provides outputs, while the participating parties change over time.

Note, in Definition 3 parties can simply "arrive in an event" by just "occurring in that event". However, w. r. t. an OTTP and real parties, this needs to be modeled more explicitly: A party will *arrive* by only *connecting* to the OTTP, i. e., establishing a private channel without sending any input. Then, in a second step (after the party has already arrived/connected) the party can now send its actual input to the OTTP. The corresponding event is then implicitly created by the OTTP itself. In other words, a party first connects to the OTTP (it arrives), then the (already) arrived party sends its input (not part of the arrival itself) to the OTTP, and then finally the OTTP creates the corresponding event, and evaluates the decision functionality. After a decision is computed, the outputs are distributed, and a new party can arrive.[12]

Definition 5 (Ideal Online TTP[13]). *Let $p_1, p_2, \ldots \in \mathcal{P}$ be an ordered sequence of parties arriving. An* ideal online TTP (OTTP) *evaluates an online functionality with decision function $\overset{\infty}{f} : \Phi \times \mathcal{E} \rightarrow \Phi \times \mathcal{D}$ as follows: The ideal OTTP stores the initial state $\phi_0$.*

1. *The i-th party $p_i^{IN}$ arrives by connecting to the OTTP, i. e., establishing a (private) communication channel only.*
2. *Party $p_i^{IN}$ sends its input $x_i$ to the OTTP.*
3. *The ideal OTTP sets the i-th event as $\sigma_i := (p_i^{IN}, x_i)$.*
4. *The ideal OTTP computes $(\phi_i, \gamma_i) := \overset{\infty}{f}(\phi_{i-1}, \sigma_i)$, where $\gamma_i := \{(p_{i,1}^{OUT}, y_{i,1}), \ldots, (p_{i,m_i}^{OUT}, y_{i,m_i})\} \in \mathcal{D}$ and $m_i \in \mathbb{N}_0$.*
5. *The ideal OTTP sends output $y_{i,k}$ to party $p_{i,k}^{OUT} \in \mathcal{P}_i^{OUT} \subseteq \mathcal{P}$ for $k \in \{1, \ldots, m_i\}$.*
6. *The ideal OTTP stores the new state $\phi_i$ and deletes the old state $\phi_{i-1}$.*

We want to stress again that we assume that the sequence of parties arriving is unknown to all parties. This implies that parties participating do not know which other parties participate. Note, this differs from other models where the participating parties are assumed to be known. In addition, a party does not know for which event $\sigma_i$ it provides input. If a party does not provide or receive any input or output, then the party does not learn that an event occurred, i. e., a party arrived or provided input. This is different to a reactive TTP, where all participating parties know when the function is evaluated. We also want to stress the fact that w. r. t. online functionalities there is a difference between a party receiving nothing, and a party receiving a string with content "empty output", as the latter one is actually an output and additional information compared to the first one, e. g., when receiving "empty output" the party learns that an event happened, which is not learned if nothing is received at all. This differs from offline and reactive functionalities: Offline functionalities are computed once and receiving nothing implies "empty output". Similarly, for reactive functionalities, as all parties provide input in each round, receiving nothing in a single round implies "empty output".

The overall goals of an ideal OTTP are to hide *which* and *when* parties arrive or leave, the *input* and *output* of each party, and the *state*. The states $\phi_i$ are only known to the ideal OTTP, except $\phi_0$.

An example execution for an online matching problem with general arrival is given in Example 1.

Example 1 (Online Matching with general arrival[14]). *The set of parties is infinite and defined as $\mathcal{P} := \{p_1, p_2, \ldots\}$. The input space is $\mathcal{X} := \{1, \ldots, 10\}$. For simplicity we assume two parties are matched if their inputs are equal. The state will store the unmatched parties and their input values. The initial state is $\phi_0 := \emptyset$.*

*Assume the first party that arrives is $p_5$, and it uses 9 as its input. The corresponding first event is then $\sigma_1 := (p_5, 9)$. As it is the first party that arrives, no match is possible. The decision is $\gamma_1 := \emptyset$, and the new state is $\phi_1 := \{(p_5, 9)\}$. The next party that arrives is $p_2$, and it uses 3 as input. The corresponding event is then $\sigma_2 := (p_2, 3)$. As no match is possible, the decision is $\gamma_2 := \emptyset$, and the new state is $\phi_2 := \{(p_2, 3), (p_5, 9)\}$. The third party that arrives is $p_1$, and it uses 3 as input. The corresponding event is then $\sigma_3 := (p_1, 3)$. Now, $p_1$ is matched with $p_2$. The decision is then $\gamma_3 := \{(p_1, p_2), (p_2, p_1)\}$ where the output of each party is the ID of the party it is matched with. The new state is then $\phi_3 := \{(p_5, 9)\}$.*

The ideal OTTP hides even the participation of parties perfectly. However, hiding the participation in the real world is hard (cf. Appendix B). We therefore introduce an additional flavor of an OTTP, where arriving, input providing, and leaving parties are broadcast, i. e., whenever a party arrives or provides input the party that arrives or provides input, the parties that are present, and the parties which are leaving are known, but the actual inputs are still private. *Broadcasting* some information means sending that information to all parties, and to the adversary.[15] In Appendix C we will introduce additional flavors for more general arrivals (including providing input multiple times).

[12]Correction: In the original version [24] the arrival of parties w. r. t. an OTTP was modeled as just "receiving an event", instead of a party arriving and then sending its input. However, this was to abstract and thus we modeled it here more explicitly. Note, we always modeled it that way w. r. t. the online SMPC protocol (cf. Protocol 2 in Section 5).

[13]Correction: In the original version [24] the OTTP received an event containing a party with its input. This was too abstract, and thus here we split it into a party first only arriving, and then in a second step just sending its input. In addition, previously a party $p_{i,k}^{\text{OUT}} \in \mathcal{P}_i^{\text{OUT}}$ occurring in the decision $\gamma_i$ obtained output if and only if $y_{i,k} \neq \perp$. However, as here now $\perp \notin \mathcal{X}$ and $\perp \notin \mathcal{Y}$, this condition is now removed.

[14]Correction: In the original version [24] we used only events in the example. Now, it is modeled as parties arriving and providing inputs.

[15]Correction: In the original version [24] we did not explicitly mention that the adversary also receives broadcast messages, which is a standard assumption in any SMPC. The broadcast messages need to be received by the adversary as well, as otherwise the simulator could obviously never simulate the messages on the bulletin board (introduced later), as it would not know which parties are present if no malicious party has arrived.

Definition 6 (OTTP with broadcast of Input Parties[16]). *An* OTTP with broadcast of input parties (OTTPI) *is an ideal OTTP that additionally behaves as follows:*

- *Whenever a party $p_i^{IN}$ arrives, the OTTPI broadcasts the arrival of party $p_i^{IN}$ before $p_i^{IN}$ can provide input.*
- *Whenever a party $p_i^{IN}$ provides input, the OTTPI broadcasts that party $p_i^{IN}$ provides input.*
- *After each decision $\gamma_i$ (and possible output distribution), the OTTPI broadcasts the set of parties present $\mathcal{P}_i$.*

## 4.3 Security Model

In the following we will define what it means for a protocol to securely realize an OTTP. Note, our definitions are heavily inspired by Goldreich [19] and Canetti [10]. However, their definitions do not allow to model an ideal OTTP where the participation itself is perfectly hidden.

The ideal OTTP does not leak which parties participate or when they provide input and thus, an adversary cannot decide which parties to corrupt. This problem is avoided by outsourcing the corruption of parties. We therefore introduce the *creator* as a new entity, which will create new parties and define their behavior. In addition, it will also decide when and which parties will be corrupted by the adversary.[17] The adversary then takes control over corrupted parties and learns their complete internal state, i. e., its input and all messages received or sent so far. We want to stress that the adversary cannot choose which parties to corrupt. We will then require a protocol to be able to deal with all valid corruption strategies of the creator, and thus essentially check every valid corruption strategy any classical adversary could have chosen.

Definition 7 (Creator). *A* creator *is an almighty entity. It is computationally unbound, non-corruptable and behaves deterministic w. r. t. its input (i. e., the content of its random tape). A creator $C$ with input $z \in \{0,1\}^*$ is denoted as $\mathrm{C}(z)$. A creator creates parties $p \in \mathcal{P}$ (including their inputs and behavior) and defines when they arrive. In addition, a creator can decide to corrupt (or un-corrupt) a party, i. e., the adversary can control that party (or loses control).[18] The creator can always see all internal states of every entity, including any TTP.*

Instead of defining security by comparing a TTP with a protocol directly, we use the idea of Canetti [10] and compare the execution of protocols only. Interacting with an OTTP is then executing a protocol that instructs the parties to send their inputs to the OTTP. This also allows the use of specialized TTPs within any protocol, e. g., an ideal TTP that implements a bulletin board. We use this approach, as it also allows comparing different OTTPs.

The *view* of an adversary $A$ in protocol $\pi$ consists of the complete internal state of all corrupted parties and of all their messages sent or received. We then define the *joint output* of an adversary $A$ and all honest participating parties for a protocol $\pi$ as the output of the adversary and the output of all (honest) parties.[19] For a creator $C$ with input $z$ and security parameter $k$ we denote this joint output as $\textsc{exec}_{\pi,A}(\mathrm{C}(z),k)$. We define $\textsc{exec}_{\pi,A,C} := \{\textsc{exec}_{\pi,A}(\mathrm{C}(z),k)\}_{z\in\{0,1\}^*,k\in\mathbb{N}}$ as the corresponding probability ensemble. Informally, a protocol $\pi$ *perfectly emulates* a protocol $\varpi$ if the joint output of $A$ for $\pi$ is perfectly indistinguishable from the joint output of a corresponding simulator $S$ for $\varpi$:

Definition 8 (Perfect Emulation[20]). *Let $C$ be a creator. A protocol $\pi$* perfectly emulates *a protocol $\varpi$ if for any adversary $A$, there exists an adversary (simulator) $S$ (polynomial in the complexity of $A$) such that*

$$\textsc{exec}_{\pi,A,C} \overset{p}{\equiv} \textsc{exec}_{\varpi,S,C}.$$

*Statistical* or *computational secure emulation* can be defined by requiring that the probability ensembles are only statistically or computationally indistinguishable. Computational secure emulation also requires the adversary to be probabilistic polynomial time.

We assume honest parties do not share their input with other parties. In addition, we assume the existence of a broadcast channel and private channels between all parties.

## 4.4 Adversary and Creator

In classical SMPC two main adversary types are considered, namely a *semi-honest* and a *malicious* adversary [19]. The semi-honest adversary follows the protocol definitions, but it stores all sent and received messages in order to deduce more information. A malicious adversary can additionally arbitrarily deviate from the protocol, e. g., send wrong messages or stop sending messages at all. The overall goal of the adversary is to either deduce more information, e. g., w. r. t. private inputs or outputs of other parties, or alter the outcome of the protocol run, i. e., the execution produces incorrect results. The semi-honest or malicious adversary can also have different "corruption powers". W. r. t. offline or reactive SMPC these are mainly *static*, *adaptive*, and *mobile* adversaries [19]: The static adversary can corrupt up to $m$ parties chosen *before* the protocol execution. An adaptive adversary can corrupt up to $m$ parties chosen *during* the protocol execution. Mobile adversaries can also corrupt up to $m$ parties during the protocol execution, but additionally can decide to corrupt a new party with the cost of losing control over an already corrupted party. In other words, the adversary can choose multiple times during the execution to corrupt up to $m$ parties chosen freely among all participating parties.

These notions of corruption can be transferred to the online setting. However, here the set of parties participating is not fixed and the adversary together with the creator determines the actual corruption. We will transfer them as follows: The number of corrupted parties and the corruption strategy (e. g., static, adaptive or mobile) is determined by defining the creator accordingly. Semi-honest and malicious behavior of corrupted parties is determined by the adversary.

[16]Correction: In the original version [24] for each event the party that provides input and set of parties present was broadcast. However, as events are now split into an "arrival" and "input providing" phase, we changed this to broadcasting the party that arrives, and broadcasting when a party provides input. In addition, we modeled the behavior more explicitly.

[17]The creator is somewhat similar to the environment in the UC-framework [10] as it creates parties and their inputs. However, here the creator will not act as a distinguisher.

[18]Note the un-corruption of a party is only important for mobile corruption strategies (cf. Section 4.4) which will not be considered here.

[19]The adversary learns everything a corrupted party learns, and thus it is not necessary to include the output of corrupted parties separately.

[20]Correction: In the original version [24] we required that the protocols $\pi$ and $\varpi$ are both probabilistic polynomial time. We removed this requirement for both protocols, as otherwise only probabilistic polynomial time algorithms can be evaluated, and thus not "any" online algorithm.

For simplicity, first consider an offline setting with $m$ semi-honest (corrupted) parties of a total of $n$ parties: The creator first creates $n$ parties and (randomly) selects $m$ parties to corrupt. The adversary controls the corrupted parties and is then defined to behave according to the protocol specification (semi-honest).

In the online setting we will assume an "honest majority" which will allow us to reuse feasibility results for offline functionalities when showing how to perfectly emulate an OTTPI in Section 5. Therefore, we will essentially consider a limit of corrupted parties that participate at every point in time, i. e., an honest majority of participating parties at all times. This means that the creator has to carefully select which parties can be corrupted at which point in time.

A *corrupted minority creator* $C_{<\frac{1}{2}}$ ensures that at each point in time the number of corrupted parties present or participating is strictly less than the number of honest parties present or participating, respectively. In addition, once a party is corrupted it cannot be un-corrupted.[21] Similarly, we define a second creator $C_{<\frac{1}{3}}$ where at most 1/3 of the parties present or participating are corrupted. For each corrupted party, the *semi-honest adversary* $A_{\text{sh}}$ learns their internal state and all their messages sent or received. The *arrival respecting malicious adversary* $A_{\text{am}}$ learns the same as the semi-honest adversary, but additionally controls the corrupted party, i. e., when to send which messages. However, $A_{\text{am}}$ will always let the corrupted party arrive as determined by the creator, and $A_{\text{am}}$ will let the corrupted party provide its (adversary controlled) input.[22] Enforcing the arrival ensures that the creator $\mathrm{C}(z)$ determines the number of corrupted parties and not the adversary by delaying the arrival. Note that this is not an actual restriction, as there always exists an input $z'$ such that creator $\mathrm{C}(z')$ lets that particular party arrive at a later point in time.

## 4.5 Perfectly Emulating Offline Functionalities

The standard definitions of security for offline and reactive functionalities with static corruption (e. g., see Goldreich [19]) are a special case of Definition 8: The creator creates the honest and corrupted parties at the very beginning and then stops generating new parties. This restricts the setting to a fixed set of parties.

Theorem 1 (Offline Feasibility Results). *For every offline functionality there exists a (SMPC) protocol perfectly emulating the corresponding offline TTP with known set of participating parties in the presence of an adversary $A_{sh}$ and a creator $C_{<\frac{1}{2}}$, or in the presence of an adversary $A_{am}$ and a creator $C_{<\frac{1}{3}}$.*

The feasibility result of Theorem 1 was proven by Ben-Or et al. [5]. Note, they did not prove security w. r. t. Definition 8 directly, but in the offline setting Definition 8 and the security definition of Ben-Or et al. [5] are essentially identical. Ben-Or et al. [5] used Shamir's secret sharing scheme [27] to share the private inputs. They showed how to securely evaluate arithmetic circuits (addition and multiplication of shared secrets), if the threshold is chosen as $\lceil \frac{1}{2}n \rceil$ in the presence of a semi-honest adversary and as $\lceil \frac{1}{3}n \rceil$ in the presence of a malicious adversary, where $n$ is the total number of parties participating.

## 4.6 Ideal Bulletin Board

In the ideal OTTP setting parties can arrive by connecting to the OTTP. However, if we replace the OTTP with a (distributed) protocol, then the parties need to know with whom to exchange messages. Therefore, we will use an ideal *bulletin board* $\mathcal{BB}$ that is *append-only* with *certified publishing* [22]: Parties can write/store messages on $\mathcal{BB}$. All messages on $\mathcal{BB}$ are readable by everyone, and can neither be altered nor deleted (append-only). Every message $(m)_p$ has an identifiable author $p$ (certified publishing).

Here, a party $p$ that arrives writes $(\texttt{arrive})_p$ on $\mathcal{BB}$. For ease of notation we assume $\mathcal{BB}$ to be *majority consistent*: Only together, a majority of parties participating during an event can write a new set of parties present $\mathcal{P}_i$ on $\mathcal{BB}$, i. e., if $p_i^{\text{IN}}$ arrives in event $\sigma_i$ and $\mathcal{P}_{i-1}$ is the set of parties already present, then a majority of them can write together $(\texttt{present}, \mathcal{P}_i)_{\mathcal{P}_{i-1} \cup \{p_i^{\text{IN}}\}}$. Note, the same functionality can be achieved by every party individually writing the set of parties present on $\mathcal{BB}$. Then, a new arriving party checks the complete history in order to find out which parties are actually present (based on the majority of messages).

We denote an append-only and majority consistent bulletin board with certified publishing in the following as $\mathcal{BB}$. A possible candidate was proposed by Heather and Lundin [22].

## 5 FEASIBILITY RESULT

In order to evaluate arbitrary online functionalities with secure SMPC protocols, the current state of the OTTPI needs to be stored among the parties. This requires that there are always parties present that can store the current state, without being able to learn it. This can be done by secret sharing the state among the parties that are present.[23] In order to allow that at any point in time at least one corrupted party can be present, we need to restrict the minimal number of parties: For an adversary $A_{\text{am}}$ with creator $C_{<\frac{1}{3}}$ we require that there are always at least $n \geq 4$ parties present, and $n \geq 3$ for $A_{\text{sh}}$ with $C_{<\frac{1}{2}}$. A larger $n$ tolerates more corrupted parties, but requires that there are always $n$ parties present. The final choice of $n$ depends on the actual application.

An OTTPI is then emulated by first "sharing" the initial state $\phi_0$ among the first $n$ parties that have arrived. A special ramp-up phase is used to evaluate the decision functionality for the first $n$ events (that would have happened). Note, that the ramp-up phase differs from the ideal model, as now every party of the first $n$ parties already knows every other party of the first $n$ parties that arrive (will provide input), before any party provided input, e. g., the first party learns who is the second party that will arrive/provide input before the first party itself hast to provide any input. After the ramp-up phase, whenever a new party arrives, the decision function is then evaluated with an SFE protocol that receives the input and the shares of the state, and computes and distributes the output. Afterwards the set of parties present is published on $\mathcal{BB}$ so that

[21] This corresponds to a static or adaptive adversary in classical SMPC.

[22] Correction: In the original version [24] the arrival respecting malicious adversary let parties arrive in the corresponding event, and thus implicitly they always provide input, too. As events are now split in arriving and providing input, we now need to explicitly enforce that all parties provide input, too.

[23] Note that there are online functionalities that cannot be implemented securely, e. g., any online functionality where at some point only one party is present, as it is impossible to store the state securely within one party.

new parties arriving know with whom to communicate. A single evaluation of the decision function is described in Protocol 1 and the protocol emulating an OTTPI is described in Protocol 2:

Protocol 1 (Evaluating the decision function for a single event). *Let the set of parties present be $\mathcal{P}_{i-1}$ as defined by $\mathcal{BB}$. Evaluate the decision functionality $\overset{\infty}{f}$ for $\sigma_i := (p_i^{IN}, x_i)$ as follows:*

(1) *The parties in $\mathcal{P}_{i-1}$ and $p_i^{IN}$ evaluate $\pi_i$, where protocol $\pi_i$ perfectly emulates the following offline functionality $f_i$:*
   (a) *Party $p_i^{IN}$ uses $x_i$ as input. The parties in $\mathcal{P}_{i-1}$ use their shares of $\phi_{i-1}$ as input.*
   (b) *Reconstruct the state $\phi_{i-1}$ from the shares, create the event $\sigma_i := (p_i^{IN}, x_i)$, and compute $(\phi_i, \gamma_i) := \overset{\infty}{f}(\phi_{i-1}, \sigma_i)$.*
   (c) *Send the outputs to each party according to $\gamma_i$. Parties not contained in $\gamma_i$ obtain the empty output $\perp$.*
   (d) *Output the set $\mathcal{P}_i$ of parties present in the new state $\phi_i$ to each party.*
   (e) *If $|\mathcal{P}_i| < n$ the execution halts. Otherwise, share the state $\phi_i$ among the parties $\mathcal{P}_i$ using Shamir's secret sharing scheme with threshold $t_i := \lceil \frac{1}{\tau} |\mathcal{P}_i| \rceil$.*

(2) *The parties in $\mathcal{P}_{i-1} \cup \{p_i^{IN}\}$ write $(\texttt{present}, \mathcal{P}_i)_{\mathcal{P}_{i-1} \cup \{p_i^{IN}\}}$ on $\mathcal{BB}$.*

Protocol 2 (Emulating an OTTPI). *An online functionality with decision function $\overset{\infty}{f}$ and initial state $\phi_0$ is evaluated as follows:*

(1) *The first $n$ parties $p_1^{IN}, \ldots, p_n^{IN}$ that arrive write $(\texttt{arrive})_{p_i^{IN}}$ for $i \in \{1, \ldots, n\}$ on $\mathcal{BB}$. All parties in $\mathcal{P}_0 := \{p_1^{IN}, \ldots, p_n^{IN}\}$ write $(\texttt{present}, \mathcal{P}_0)_{\mathcal{P}_0}$ on $\mathcal{BB}$.*
(2) *The parties in $\mathcal{P}_0$ share the initial state $\phi_0$ as follows:*
   (a) *Set $t_0 := \lceil \frac{1}{\tau} |\mathcal{P}_0| \rceil$ and $g(x) := \phi_0 + \sum_{j=1}^{t_0 - 1} x^j \mod q$, where $q$ is a sufficiently large prime.*
   (b) *Party $p_i^{IN}$ with $i \in \{1, \ldots, n\}$ uses $g(i)$ as its share of $\phi_0$.*
(3) *Evaluate the decision functionality $\overset{\infty}{f}$ for $i \in \{1, \ldots, n\}$:*
   (a) *Execute Protocol 1 for event $\sigma_i := (p_i^{IN}, x_i)$ with the following modifications:*
      (i) *Instead of Step 1e): If $|\mathcal{P}_i| \neq i$, the execution halts. Otherwise share $\phi_i$ among $\mathcal{P}_0$ with $t_i := t_0$.*
      (ii) *Instead of Step 2: The parties write $(\texttt{present}, \mathcal{P}_0)_{\mathcal{P}_0}$ on $\mathcal{BB}$, i. e., $\mathcal{P}_i := \mathcal{P}_0$ on $\mathcal{BB}$.*
   (b) *If Protocol 1 has halted early, then the execution stops.*
(4) *Set $i := n + 1$.*
(5) *Wait until a party $p_i^{IN}$ with input $x_i$ arrives, i. e., $p_i^{IN}$ has written $(\texttt{arrive})_{p_i^{IN}}$ on $\mathcal{BB}$.*
(6) *Evaluate the decision functionality $\overset{\infty}{f}$ with Protocol 1.*
(7) *If Protocol 1 has halted early ($|\mathcal{P}_i| < n$), then the execution stops. Otherwise, set $i := i + 1$ and got to Step 5.*

Note, the initial state $\phi_0$ in Protocol 2 is modeled as a single value instead of a vector for clarity of notation. However, in an actual application (possibly) the initial state, but especially any consecutive states (cf. Protocol 1) are likely to be vectors.[24] Therefore, the sharing of any state has to be understood as the sharing of the individual entries of a state-vector, instead of a single state-value. The same holds for the input and output of parties.[25]

In Protocol 1, parties not included in the decision obtain $\perp$ as output. As by definition $\perp$ does not occur in the output space, i. e., $\perp \notin \mathcal{Y}$, this needs to be considered when actually implementing a corresponding SMPC protocol. One possible approach is to output an additional bit, which indicates whether the output is actually $\perp$: For simplicity assume the output is a single value. If a party should receive an actual output (it occurred in the decision), then the additional bit is set to 1, and the output value is set to the actual output. If a party should not receive any output (it should receive $\perp$), then the additional bit is set to 0 and the output value itself is also set to 0. This approach does not only encode $\perp$, but also allows to distinguish between the output 0 and $\perp$. In addition, this approach prevents a (malicious) party from providing $\perp$ as input: If one would only use a single value 0 to represent $\perp$, then a (malicious) party could use that also as input, as typically the input and output space of offline SMPC protocols are the same, e. g., as in the approach by Ben-Or et al. [5]. However, by having the additional bit not being a part of the input phase itself, a party cannot provide $\perp$ as input.[26]

Theorem 2. *Protocol 2 perfectly emulates an OTTPI for any online functionality with decision functionality $\overset{\infty}{f}$ in the presence of an adversary $A_{am}$ and a creator $C_{<\frac{1}{3}}$ with $\tau := 3$, or in the presence of an adversary $A_{sh}$ and a creator $C_{<\frac{1}{2}}$ with $\tau := 2$. This holds only if $n \in \mathbb{N}$ and $n > \tau$, and initially at least $n$ parties arrive first without providing input (while their arrival is broadcast by the OTTPI), and then those $n$ parties provide their inputs, and then at least $n$ parties are always present.*[27]

Note, the underlying (offline) SMPC protocol requires that at least $n$ parties initially arrive before any computation can be performed, e. g., protocols based on Shamir's secret scheme typically require at least 3 parties to perform computations while having at least one corrupted party [5]. Therefore, initially $n$ parties have to arrive (by writing "arrive" on $\mathcal{BB}$) first without providing any input (not yet possible), and then after $n$ parties arrived, they can provide their input. Hence, in the ideal model initially $n$ parties have to arrive first without providing input, too. And then after $n$ parties arrived, they can also provide their input. This is, e. g., required if a malicious adversary is present and $n \geq 5$, as then a malicious party arriving as the fourth party can choose its input based on knowing who the fifth party is.[28]

Proof. First, observe that the threshold $t_i$ used to share the state $\phi_i$ is chosen such that at any point in time no set of corrupted parties can reconstruct the state, alter the state, or prevent reconstruction for $C_{<\frac{1}{\tau}}$: The honest parties have always enough shares (honest

[24]Note, the initial state can always be a fixed value, e. g., $\phi_0 := 0$, as the initial state can be encoded directly in the decision functionality $\overset{\infty}{f}$ itself. In addition, we think that for many application the initial state is even empty, as no party has arrived or provided input yet, e. g., for online matching with general arrival.

[25]Correction: In the original version [24] we did not clarify that the state and input should be seen as vectors.

[26]Correction: In the original version [24] $\perp$ was part of the input an output space. However, here it is removed, and thus we additionally describe an approach how to encode $\perp$.

[27]Correction: In the original version [24] we just stated that "initially at least $n$ parties arrive", but the intended meaning was unclear. This is now corrected by specifying this more precisely.

[28]Extension: We added this paragraph to describe why we require that initially $n$ parties have to arrive first without providing input.

majority), and the corrupted parties always lack at least one share (choice of the threshold).

Second, according to Theorem 1 there exists a protocol $\pi_i$ that perfectly emulates the offline functionality $f_i$ in Protocol 1 for $A_{\text{am}}$ and $C_{<\frac{1}{3}}$, or $A_{\text{sh}}$ and $C_{<\frac{1}{2}}$, respectively. Thus, there exists a corresponding simulator $S'_i$ for $f_i$. This includes that simulator $S'_i$ can extract the inputs of corrupted parties in order to send them to the OTTPI. In addition, the set of parties present is broadcast after every decision (and possible output distribution) by the OTTPI. Therefore, the simulator $S'_i$ knows when a decision was computed and when (possibly) corresponding outputs were distributed. Thus, if corrupted parties do not obtain output from the OTTPI, then the simulator $S'_i$ can ensure that they obtain $\perp$. As the set of parties present is broadcast after each decision, the simulator $S'_i$ can also ensure that $\pi_i$ outputs the same set of parties present. The consistency of the shares provided by the parties in $\mathcal{P}_{i-1}$ is (implicitly) checked by $f_i$ when it reconstructs the state $\phi_{i-1}$. This also covers the evaluation of the first $n$ events, especially of $\sigma_1$. Note, the shares of the initial state is shared with a fixed polynomial, and the shares of consecutive states are generated by $f_i$. Therefore the shares of honest parties will always be correct (similar as if a TTP would share a secret).

Finally, the main simulator $S$ has to simulate the messages on $\mathcal{BB}$ and call the corresponding simulator $S'_i$ multiple times: The messages on $\mathcal{BB}$ are trivial to simulate by the simulator $S$ as whenever a party arrives or provides inputs, the OTTPI broadcasts the party which arrives or provides input. In addition, after each decision the OTTPI broadcasts the set of parties present, too. Especially the initially arriving $n$ parties are broadcast before they provide any input. In particular, initially $n$ parties arrive, and their arrival is broadcast by the OTTPI. Note, no party can provide input, unless all $n$ parties have arrived. The simulator $S$ writes the corresponding arrive messages on $\mathcal{BB}$ as soon as they are broadcast. After $n$ arrive messages are written on $\mathcal{BB}$ (and thus corresponding broadcasts), the simulator $S$ writes the set of parties present $\mathcal{P}_0$ on $\mathcal{BB}$. Next, the initially arriving $n$ parties provide their input one after another, where each sending of input is broadcast by the OTTPI. Each broadcast of a party providing input, and each broadcast of the set of parties present after a decision (and possible output distribution), corresponds to the evaluation of $\overset{\infty}{f}$ and corresponding output distribution. By definition, the offline functionality $f_i$ computes the decision functionality $\overset{\infty}{f}$ correctly. Therefore, the simulator $S$ can simply call the corresponding simulator $S'_i$. In addition, for the first $n$ decisions the simulator $S$ will always write $(\texttt{present}, \mathcal{P}_0)$ on $\mathcal{BB}$.

After the ramp-up phase, the "normal" executions starts. A new party that arrives is broadcast by the OTTPI, and thus the simulator writes the corresponding arrive message on $\mathcal{BB}$. When the newly arrived party provides input, which again is broadcast, the simulator $S$ calls again the corresponding simulator $S'_i$ to simulate the corresponding evaluation of the decision functionality. As the set of parties present is broadcast after each decision, the simulator $S$ can write the corresponding set of parties present on $\mathcal{BB}$. If at some point the number of parties present is below $n$, or during the ramp-up phase $|\mathcal{P}_i| \neq i$, the complete execution halts. As the simulation of $S_i$ is perfect, then so is the combined simulation done by $S$.[29] □

Remark 1 (Requirement of Memory). *Recall that the state has to store the set of parties in order to compute the decision function. However, if we assume an OTTPI, then the parties present are known and can be hard coded in a modified protocol $\pi'_i$. Therefore, a state needs to be stored only if some values need to be available in multiple events. Hence, some online functionalities can be realized without the need to share a state. Recall online matching with general arrival (cf. Example 1), and assume an OTTPI with a semi-honest adversary. Whenever a new party arrives with their input, the previously unmatched parties will use their original input as input for $\pi_i$ so that no state needs to be shared in this case. This does not work with malicious adversaries, as they could easily alter their input, so the state needs to be shared in this case.*

Remark 2 (Malicious Parties aborting). *The OTTPI and Protocol 2 can deal with malicious parties that abort as long as they have provided their input. We do not exclude malicious parties once they are detected: A* rational *malicious party will not behave maliciously during the computation, as it otherwise would be detected. In the worst case, it will only change its input, which cannot be detected. However, if one wants to blame a malicious party $p'$, then a majority can easily write $(malicious, p')_{\mathcal{P}_{i-1} \cup \{p_i^{IN}\}}$ on $\mathcal{BB}$ and then $p'$ is implicitly excluded in any further communication.*

Remark 3 (Computational Security). *Cramer et al. [13] showed that for every offline functionality there exists a corresponding computationally secure SMPC protocol. In their constructions they use a homomorphic threshold cryptosystem. This allows tolerating $A_{am}$ with $C_{<\frac{1}{2}}$, or $A_{sh}$ with $C_{<1}$ (at least one honest party). Protocols 1 and 2 and the corresponding proof can be easily adapted to use Cramer et al. [13] scheme by storing the state as ciphertext.*

Remark 4 (Mobile Adversaries). *In order to keep our model and construction simple, we did not consider mobile adversaries (proactive security). However, our construction in Protocols 1 and 2 could be adapted roughly as follows: First, use the proactive secure building blocks introduced by Eldefrawy et al. [16] (instead of Shamir's secret sharing scheme). Second, incorporate the refreshing of shares at suitable steps, e. g., multiple times when parties wait for new parties to arrive, and during the execution of Protocol 1.*

## 6 LIMITS OF THE OTTPI

The ideal OTTP is essentially the *holy grail* w. r. t. privacy and security, as no party knows which or when other parties participate or provide input. Not even the number of participating parties is known.[30] In classical SMPC parties are assumed to know which other parties participate. It is therefore important to see the results in this section not as a contradiction to our feasibility result in Section 5, but as a motivation that more research is required w. r. t. hiding participating parties in general. This is also desirable for classical SMPC: Assume a simple offline function computing the

[29]Extension: We added more details to the proof, especially w. r. t. the simulator's behavior.
[30]An imperfect comparison is to assume that blind and deaf parties participate.

average $M$ of the parties (positive) inputs.[31] Knowing that $n$ parties participate leaks that the input of each party is at most $M \cdot n$. However, this is not leaked if we do not know how many parties participated.

An OTTPI leaks whether a party participates, and the order in which parties arrive and leave. An ideal OTTP hides both information. However, knowing the order in which parties arrive can leak additional private information. In order to capture that leakage, we compare an OTTPI and an ideal OTTP where in both settings the parties that have participated are known after the execution, but only in the OTTPI setting the order in which the parties arrive and leave is known. Intuitively, an online functionality provides *privacy with public arrival* if one cannot deduce additional information from knowing the order in which certain parties arrived or left:

Definition 9 (Privacy with Public Arrival[32]). *An online functionality guarantees* privacy with public arrival *if for any finite set $\mathcal{P} := \{p_1, \dots, p_m\}$ of parties arriving uniformly at random, the corresponding OTTPI perfectly emulates the corresponding ideal OTTP if after the last decision $\gamma_m$ is distributed, the unordered set $\mathcal{P}$ of all parties that participated is published.*

Note, that Definition 9 implicitly allows the simulator (interacting with the ideal OTTP) to insert into its view (after $\mathcal{P}$ is published) a (random) order of how the parties might have arrived: The simulator does not learn the order in which parties have actually arrived (if the functionality does not leak that explicitly), and thus only learns which parties have arrived, i. e., the simulator learns the unordered set $\mathcal{P}$. However, the adversary (interacting with the OTTPI) learns the order in which parties arrive, and thus will include it in its view. Security (cf. Definition 8) is defined by comparing the joint outputs of the adversary and simulator (both including the output of honest parties to guarantee correctness). Therefore, the simulator has to select and insert (in retrospect) into its view an order of how the parties could have arrived such that the corresponding probability ensembles are indistinguishable, i. e., they have the same distribution. If during the execution nothing can be learned from the order in which parties arrived, then a (uniformly at) random selected order of parties arriving by the simulator (respecting the order of corrupted parties arriving and inserted in retrospect into its view) is perfectly indistinguishable from the parties actually arriving uniformly at random, as in both settings the order of parties arriving is uniformly at random, and thus the corresponding probability ensembles have the same distribution. However, if during the execution additional information can be learned from the actual order in which the parties arrived, this "trick" will no longer work, and thus the views will be distinguishable.[33]

In the following we will show that there are multiple commonly used online (matching) algorithms that do not guarantee privacy with public arrival.

[31]Correction: In the original version [24] we did not explicitly mention that the leaked information can only be obtained if we assume that all inputs are positive.

[32]Correction: In the original version [24] the definition was based on events. We changed it here to parties arriving and decisions, as events are now split into "arriving" and "providing input". In addition, we now explicitly mention that the parties arrive uniformly at random.

[33]Extension: We added this paragraph as additional explanation of Definition 9.

Proposition 1. *The GREEDY algorithm*[34] *for online matching with general arrival does not guarantee privacy with public arrival.*

Proof. Recall, a party leaves only if it is matched. As soon as two honest parties leave in the OTTPI model, we learn that these two parties are matched. However, the simulator can only guess with probability 1/2 whether an honest party is matched or not. □

Another example is *online maximum weighted bipartite matching* [23]: The problem consists of $n$ servers present from the very beginning and $n$ requests arrive one after another. The $n$ servers are publicly known, and never leave the computation. Each request contains an ordered list of the servers reflecting the preference for each server, where the first entry has the highest preference value of $n$ and the last entry has the lowest preference value of 1. Whenever a request arrives, the online algorithm has to decide immediately with which server the request will be matched. If a server is matched once, it cannot be matched again. In addition, any decision made cannot be changed later. The goal is to match all requests such that the sum of the preference values of all matches is maximized. The optimal solution is to greedily match the request with its highest preference server that is still unmatched [23].

Proposition 2. *The GREEDY algorithm for online maximum weighted bipartite matching does not guarantee privacy with public arrival.*

Proof. Assume a (corrupted) server is matched with the first request. In the OTTPI model the server knows that it is matched with the first request, and thus learns that the request has given it the highest priority. In the ideal OTTP model the server does not know whether it is matched with the first, the $i$-th or even the last request, and thus does not know whether it is matched due to being preferred to others, or because it is the last unmatched server. □

Even stronger results hold for online bipartite matching, where similar to online maximum weighted bipartite matching requests provide preferences for servers. However, their preferences are binary for each server indicating whether they are willing to be matched with that server or not. A request is not matched if there is no server available that the request is willing to be matched with.

Proposition 3. *No online bipartite matching algorithm in which a request is always matched if there is a suitable server available, i. e., the server is unmatched and the request is willing to be matched with that server, guarantees privacy with public arrival.*

Proof. Assume the first request $R_1$ is matched with server $S_1$. Now assume (corrupted) request $R_2$ arrives and $R_2$ is willing to be matched only with $S_1$. As $S_1$ is already matched $R_2$ cannot be matched. In the OTTPI model, $R_2$ knows that it is the second request. As it is unmatched, it can immediately deduce that $R_1$ is matched with $S_1$. In the ideal OTTP model, $R_2$ does not know that it is the $i$-th request, and thus learns only that $S_1$ is already matched. □

Another simple online matching algorithm is *RANDOM*, i. e., for online bipartite matching, whenever a request arrives, it is randomly matched with any available server disregarding the preferences of

[34]As soon as two parties can be matched they will be matched.

the request. It seems like RANDOM could guarantee privacy with public arrival, but even RANDOM does not:

Proposition 4. *The RANDOM algorithm does not guarantee privacy with public arrival.*

Proof. Each match is completely random, and the requests do not provide any input. Hence, nothing can be learned about their inputs. The first $n$ request will be matched with the $n$ servers. All requests that arrive later cannot be matched anymore, and they learn that all $n$ servers are already matched. In the OTTPI model, an adversary obviously learns that the first $n$ requests are "matched", and the output for all later requests is exactly "not matched". However, in the OTTP model, a participating request learns either it is matched, or if is unmatched and at least $n$ other requests are matched. It does not learn which requests are matched or unmatched. □

However, if we restrict the amount of possible events, then RANDOM can guarantee privacy with public arrival:

Proposition 5. *The RANDOM algorithm guarantees privacy with public arrival if the execution is stopped immediately after all servers are matched.*

Proof. W.l.o.g. assume a total of $n$ servers. The execution is halted after exactly $n$ requests have arrived and are matched. Now assume that after decision $\gamma_i$ with $i \in \{1, \ldots, n\}$ the set of all participating parties is published. The view w. r. t. corrupted parties is trivial to simulate as they are identically in both settings: The party connects to the simulator, the simulator connects as corresponding party to the ideal OTTP, and returns the corresponding outputs of the ideal OTTP.[35] All other requests are matched by definition, and thus trivial to simulate. The order of the request in which they arrive can be chosen uniformly at random respecting the order of matched corrupted requests. As there is no prior knowledge on when requests arrive, except that they arrive uniformly at random, this is perfectly indistinguishable. □

Another positive example is bipartite matching where every request is always matched (ALWAYS) with the best fitting server, i. e., a server can be matched multiple times with different requests.

Proposition 6. *The ALWAYS algorithm guarantees privacy with public arrival with an adversary $A_{sh}$, but not with an adversary $A_{am}$.*

Proof. Note, each match depends solely on the initial input of the server and the input of a request. As no server can get occupied, a request will always be matched with the same server independent of when it arrives, as long as the input is the same. Again, corrupted parties are trivial to simulate as their inputs and outputs in both settings are identical. The order of the other requests arriving can be chosen uniformly at random respecting the order of matched corrupted requests. This is perfectly indistinguishable for a semi-honest adversary, but not for a malicious adversary. A malicious adversary can store some information about previous requests by enforcing a certain matching, e. g., two servers $s_1, s_2$ are corrupted and at least one request $r$. By letting request $r$ be matched with either $s_1$ or $s_2$ we can store information about whether exactly one ($s_1$) or more requests arrived ($s_2$). As the simulator can only guess how many requests already arrived, it is likely that the simulator guesses wrong. Thus, the joint outputs are not identical. □

[35] Correction: In the original version [24] the simulator forwarded inputs, but there are no inputs. Therefore, we removed that, and included instead the simulation of the connection establishment.

## 7 ADDITIONAL RESULTS

We briefly discuss the setting with a dishonest majority in Appendix A, and argue in Appendix B why the arrival time can in general not easily be hidden. In Appendix C we will allow parties to arrive, leave and provide input arbitrarily, and therefore introduce the OTTPE and OTTPP, and briefly present the corresponding feasibility results.

## 8 CONCLUSION

In this paper we proposed several new models for TTPs that can evaluate online functionalities. We described two main types: The ideal OTTP where the arrival of the parties is private, and the OTTPI where it is broadcast. We also extended the model to allow parties to arrive, leave and provide input arbitrarily, and introduced additional models, namely the OTTPE and OTTPP in Appendix C. We showed that for any online functionality there exists a protocol in the OTTPI (and OTTPE/OTTPP) setting that perfectly emulates the online functionality as long as there are always at least three parties present. As the arrival of parties cannot be easily hidden with reasonable assumptions (cf. Appendix B), only the OTTPI (and OTTPE/OTTPP) model seem to be real world applicable.

In the future, we plan to investigate further how to provide better privacy than what can be achieved by an OTTPI, while being easier to realize than a protocol secure in the ideal OTTP model.

## ACKNOWLEDGMENTS

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 2236/1.

## REFERENCES

[1] Balamurugan Anandan and Chris Clifton. 2017. Secure minimum weighted bipartite matching. In *Conference on Dependable and Secure Computing*. 60–67.
[2] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P Smart, and Rebecca N Wright. 2018. From Keys to Databases—Real-World Applications of Secure Multi-Party Computation. *Comput. J.* (2018), 1749–1771.
[3] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. 2015. Communication-Optimal Proactive Secret Sharing for Dynamic Groups. In *Applied Cryptography and Network Security*. Springer, 23–41.
[4] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology - CRYPTO*. Springer, 420–432.
[5] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems For Non-Cryptographic Fault-Tolerant Distributed Computation. In *Symposium on Theory of Computing*. ACM, 1–10.
[6] Marina Blanton and Siddharth Saraph. 2014. Secure and Oblivious Maximum Bipartite Matching Size Algorithm with Applications to Secure Fingerprint Identification.
[7] Dan Bogdanov. 2013. *Sharemind: programmable secure computations with practical applications*. Ph.D. Dissertation. University of Tartu, Estonia.
[8] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* 13 (2000), 143–202.
[9] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *International Conference on Cluster Computing*. 136–145.
[10] Ran Canetti. 2020. Universally Composable Security. *J. ACM* 67 (2020), 28:1–28:94.
[11] David Chaum, Claude Crépeau, and Ivan Damgard. 1988. Multiparty unconditionally secure protocols. In *Symposium on Theory of Computing*. ACM, 11–19.
[12] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. 2020. Fluid MPC: Secure Multiparty Computation with Dynamic Participants. Cryptology ePrint Archive, Report 2020/754.

[13] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. 2000. Multiparty Computation from Threshold Homomorphic Encryption. Cryptology ePrint Archive, Report 2000/055.

[14] Ivan Damgård and Jesper Buus Nielsen. 2003. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In *Advances in Cryptology - CRYPTO*. Springer, 247–264.

[15] Yvo Desmedt and Sushil Jajodia. 1994. *Redistributing secret shares to new access structures and its applications.* Technical Report ISSE TR-97-01. George Mason University.

[16] Karim Eldefrawy, Seoyeon Hwang, Rafail Ostrovsky, and Moti Yung. 2020. Communication-Efficient (Proactive) Secure Computation for Dynamic General Adversary Structures and Dynamic Groups. In *Security and Cryptography for Networks*. Springer, 108–129.

[17] Amos Fiat and Gerhard Woeginger (Eds.). 1998. *Online Algorithms: The State of the Art.* Springer.

[18] Buddhima Gamlath, Michael Kapralov, Andreas Maggiori, Ola Svensson, and David Wajc. 2019. Online Matching with General Arrivals. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. 26–37.

[19] Oded Goldreich. 2004. *Foundations of Cryptography: Basic Applications.* Cambridge University Press.

[20] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play ANY mental game. In *Symposium on Theory of Computing*. ACM, 218–229.

[21] Philippe Golle. 2006. A Private Stable Matching Algorithm. In *International Conference on Financial Cryptography and Data Security*. Springer, 65–80.

[22] James Heather and David Lundin. 2009. The Append-Only Web Bulletin Board. In *Formal Aspects in Security and Trust.* Springer, 242–256.

[23] Bala Kalyanasundaram and Kirk Pruhs. 1993. Online Weighted Matching. *Journal of Algorithms* 14 (1993), 478–488.

[24] Andreas Klinger and Ulrike Meyer. 2021. Towards Secure Evaluation of Online Functionalities. In *The 16th International Conference on Availability, Reliability and Security (ARES 2021)*. Association for Computing Machinery.

[25] Ming Li, Ning Cao, Shucheng Yu, and Wenjing Lou. 2011. FindU: Privacy-preserving personal profile matching in mobile social networks. In *International Conference on Computer Communications.* 2435–2443.

[26] Tal Rabin and Michael Ben-Or. 1989. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In *Symposium on Theory of Computing*. ACM, 73–85.

[27] Adi Shamir. 19799. How to Share a Secret. *Commun. ACM* 22 (19799), 612–613.

[28] Stefan Wüller, Michael Vu, Ulrike Meyer, and Susanne Wetzel. 2017. Using Secure Graph Algorithms for the Privacy-Preserving Identification of Optimal Bartering Opportunities. In *Workshop on Privacy in the Electronic Society.* ACM, 123–132.

[29] Andrew C. C. Yao. 1986. How to generate and exchange secrets. In *Symposium on Foundations of Computer Science.* 162–167.

## A DISHONEST MAJORITY

In Section 5 we assumed to always have a majority of honest non-corrupted parties. Another common assumption in classical SMPC is to assume a dishonest majority [19], both with semi-honest and malicious adversaries, i. e., in the worst case only one party is honest. Any offline functionality can be realized computationally secure if an early abort is allowed [19]. In the online setting assuming a dishonest majority requires dealing with situations where at some point only corrupted parties are present, i. e., $C_{\leq 1}$: An honest party arriving has to assume that after it arrived all other parties are dishonest. Therefore, the parties that were present before the honest party arrived and the parties that are present after the honest parties leaves are all corrupted.

Proposition 7. *There are online functionalities that cannot be emulated even computationally secure without the help of an OTTP (or OTTPI) in the presence of an adversary $A_{sh}$ and a creator $C_{\leq 1}$.*

Proof. Consider matching where each input is allowed to be matched at most four times. Obviously the state has to contain some information w. r. t. the inputs, e. g., a counter for each possible input. Assuming no TTP, then only the parties themselves perform the computation and store the state somehow (shared). How this is done exactly is not important, but in order to use the state, the parties present need to be able to recover the current state.

Now assume that currently $n$ corrupted parties are present. Then, a new (honest) party arrives, provides its input, is immediately matched, and leaves again. This means that before and after the honest party is present, the adversary learns the state and thus the input of the honest party. □

A consequence of Proposition 7 is that every newly arriving party has to assume that the previous state is either known to the adversary or even altered. This essentially implies that online functionalities that require a state cannot be emulated securely by any protocol without the help of a TTP. However, if we restrict the type of online functionalities that can be evaluated such that inputs of parties can only be considered during their presence, then we can allow a dishonest majority.

Theorem 3. *An OTTPI for an online functionality can be emulated computationally secure by an SMPC protocol in the presence of an adversary $A_{sh}$ and a creator $C_{\leq 1}$ if each state $\phi_i \in \Phi$ depends solely on the inputs and outputs of the parties present in $\mathcal{P}_i$.*

Proof. Note, we will present only the main idea, without the technical details and restrictions that have already been introduced by others in classical SMPC and are required for offline or reactive SMPC with dishonest majority.

The idea is to ensure that the contribution of the inputs and outputs of a single party to the current state lasts only as long as it is present. This way each party itself can prevent reconstruction of their own input.

W.l.o.g. assume that all inputs and outputs of all parties present in $\mathcal{P}_i$ are stored in $\phi_i$. The state $\phi_i$ is organized as a database, where each entry $d_j$ corresponds to the inputs and outputs of party $p_j$. Each entry is shared among all parties present such that all parties need to cooperate in order to reconstruct the state, i. e., $t_i := |\mathcal{P}_i|$. If a new party arrives, the threshold is increased accordingly.[36] If a party $p_j$ leaves, then the threshold of all database entries $d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_{|\mathcal{P}_i|}$ is decreased. The threshold of entry $d_j$ is not touched, and thus becomes non-recoverable as soon as $p_j$ has deleted its share. □

Note, online matching with general arrival (cf. Example 1) is still possible with this approach.

The above approach only works with malicious adversaries if we allow them to modify the current state: As soon as no honest party is present, the malicious adversaries can change the state (not possible with an OTTPI). Hence, a new party arriving must assume that the state is completely controlled by the malicious parties and thus effectively require that no state is used, i. e., just use inputs of parties, and no state at all.

However, assuming a dishonest majority where the corrupted parties are split between two non-colluding adversaries, allows again to securely evaluate any online functionality: Essentially the idea is to choose the threshold for sharing the state maximal and evaluate the decision functionality with protocols secure against classical dishonest majority. As the adversaries do not collude, the state can never be reconstructed or changed by a single adversary.

[36]Note, that the threshold of Shamir's secret sharing scheme can be increased and decreased without secret reconstruction as shown by Desmedt and Jajodia in [15]: Each party shares their current share with all other parties. Then, the parties can compute locally a new share from the received shares.

## B HIDING THE ARRIVAL OF PARTIES

Section 6 shows that it is highly desirable to hide the arrival of the parties, i. e., hide that a party is the *i*-th party to arrive. If we can hide the arrival of parties, then we are able to securely simulate the ideal OTTP. However, realizing this in the real world poses multiple problems.

First, the ideal OTTP effectively hides even the presence of channels, i. e., an adversary checking all outgoing connections of all possible parties cannot observe whether a party sends messages to the ideal OTTP or not. We do not know whether this is possible to achieve (perfectly) in the real world. However, one can image an adversary with limited powers that cannot observe the existence of channels, and can only control corrupted parties. E.g. an internet service provider can check if messages are sent, but a "normal" hacker might be only able to corrupt certain parties without observing the channels of all possible parties.

In order to hide the arrival itself, the participating parties have to be at least anonymous, as otherwise it is trivially observable which party arrives. If we assume only semi-honest adversaries, then anonymous participation is achievable: All parties perform the computation anonymously and follow the protocol. However, if an adversary can be completely anonymous, the intent of being "semi-honest" seems unrealistic, as the adversary does not have to fear any consequences if he is blamed for cheating. If we allow malicious adversaries, they can easily add new parties and break the security. Therefore, parties need to be restricted such that they can arrive at most once. How to actually realize such a check in a privacy preserving manner is beyond the scope of this paper. For now, we just assume that this is possible, which leaves the problem of hiding how many parties are currently participating.

One at first glance reasonable approach (that will fail in the end) to hide the arrival of the parties is to introduce *fake parties*. A real party is an actual party in $\mathcal{P}$ participating, i. e., a party that participates also in the OTTP model. A fake party is only present in an actual protocol execution and acts like a real party, i. e., it arrives, provides input and maybe leaves. However, a fake party is not allowed to change the outcome of an evaluation, e. g., a fake party can never be matched with a real party, as then the real party would be unavailable.

Note, here we ignore the fact that fake parties introduced by real parties are somewhat contradicting to a party arriving at most once at the same time. However, we only want to show that even introducing fake parties does not solve the problem, and we therefore simply assume that the decision function will somehow distinguish between real and fake parties: First, introducing fake parties will never be able to perfectly hide the arrival of parties. The fake parties need to be introduced by the participating parties themselves following some strategy. Such a strategy is in the most general form the introduction of 0 to $M$ fake parties at certain points in time. We can safely assume an upper limit if we want to guarantee at least some efficiency. It is possible that all participating parties decide not to introduce any fake party at all and the arrival of parties is no longer masked. Note that requiring each party to always introduce a minimal amount $m$ of fake parties leads to the same result, e. g., in online maximum weighted bipartite matching the first matched server can learn that it is matched with the first party if it is matched with the $((n+1)m+1)$-th party, i. e., all parties decide to introduce only $m$ fake parties, where $n$ is the number of servers.

Second, even if we only try to achieve computational security (cf. Remark 3) with a semi-honest adversary, then the computational overhead would become computationally infeasible: Reconsider the first argument when all parties decide to introduce 0 fake parties. In order to make the probability for such an event negligible, we have to introduce exponentially many fake parties.[37]

As for now it seems impossible to hide the arrival of the parties in general.

## C EXTENSIONS

In the previous sections we restricted each party to provide input only once, namely when they first arrive. In addition, in a single event only a single party could provide input.

In this section, we remove these restrictions and thus allow to evaluate any online functionality. First, we will allow multiple parties to provide input at the same time. Second, we will additionally allow each party to provide input in multiple events.

Note, we will discuss and proof these extensions in the following only briefly, as they are rather straight forward. We include them only for completeness and omitted them in the main part of the paper for sake of clarity in the notations.

### C.1 Multiple Parties per Event

A classical extension to online algorithms where parties arrive one at a time, is to allow batch processing [17], i. e., multiple parties arrive and provide their input at the same time.

All of our results can easily be transferred to this setting with some minor modifications: The event space will be defined as $\mathcal{E} := \{\{(p^{\text{IN}}_{j_1}, x_{j_1}), \ldots, (p^{\text{IN}}_{j_\ell}, x_{j_\ell})\} \in 2^{\mathcal{P}\times\mathcal{X}} \mid \ell \in \mathbb{N}$ and all $p^{\text{IN}}_{j_1}, \ldots, p^{\text{IN}}_{j_\ell}$ are pairwise different$\}$. The events used for evaluating the online functionality are then $\sigma_i := \{(p^{\text{IN}}_{i,1}, x_{i,1}), \ldots, (p^{\text{IN}}_{i,n_i}, x_{i,n_i})\} \in \mathcal{E}$, where $x_{i,j}$ is the input of party $p^{\text{IN}}_{i,j} \in \mathcal{P}^{\text{IN}}_i \subseteq \mathcal{P}$ and $n_i \in \mathbb{N}$. We therefore define $\mathcal{P}^{\text{IN}}_i \subseteq \mathcal{P}$ as the set of all parties that arrive at the same time (for event $\sigma_i$).

The modification of the OTTPs and Protocols 1 and 2 is straight forward by exchanging $p^{\text{IN}}_i$ with $\mathcal{P}^{\text{IN}}_i$. Note, that again the parties in $\mathcal{P}^{\text{IN}}_i$ arrive first without input, and then in a second step, after all parties in $\mathcal{P}^{\text{IN}}_i$ arrived, they then provide their input. And the OTTPI broadcasts every arriving party before any party can provide input. In order to allow parties to arrive "at the same time" one needs to introduce some time window in which parties can arrive, e. g., all parties that arrived during one hour are considered for $\mathcal{P}^{\text{IN}}_i$. The exact time window depends heavily on the actual application. The parties that arrived during this time window can then provide their inputs. The event is again implicitly created, and the decision function can be evaluated. Theorem 2 still holds, and the new proof is analogous (and thus not repeated). Note, the ramp-up phase

[37] Correction: In the original version [24] we used as additionally third argument an online functionality that we thought supports that hiding of parties is sometimes even impossible. As example we used an auction, where a higher bid is immediately broadcast after a party has overbid the previous highest bid. However, at that time we did not notice that the example is essentially integrating the behavior of an OTTPI in the online function itself. Therefore, the broadcast happens in the ideal model, too. Hence, the provided example was wrong, and is now removed.

has to consider also the corresponding time windows in order to evaluate the correct corresponding events (which parties arrived for which event). However, if in the first event already $n$ parties arrive, then the ramp-up phase can be very short. Note, the results discussed in Section 6 still apply, as they are a strict subset of the new setting.[38]

## C.2 Provide Input in multiple Events

Next, we allow each party to provide input multiple times, i. e., a party can provide input not only when it arrives. In other words, we remove the restriction that each party occurs at most in one event. This includes that the set of parties participating can stay unchanged over multiple events, i. e., an event happens without a new party arriving. This can be modeled with two time windows: In the first time window parties can still only arrive, and no party can provide any input. However, in the following second time window any party that arrived during the first time window, and any party that is already present can provide input, but no party can arrive. If no party arrived and no input was provided, then no event will be created, and the decision function will not be evaluated. If at least one party arrived, or at least one party provided input, then an event will be created, and the decision function will be evaluated.[39] The OTTPI would leak which party sends multiple inputs, thus we introduce a new OTTP version that partially hides which party provides input.

Definition 10 (OTTP with Event Broadcast[40]). *An* OTTP with event broadcast (OTTPE) *is an ideal OTTP that additionally behaves as follows:*

- *Whenever one or multiple parties arrive, the OTTPE broadcasts the parties that arrive before any party can provide input.*
- *Whenever at least one party provided input (i. e., when an event would happen), the OTTPE broadcasts that an event happens.*
- *After each decision $\gamma_i$ (and possible output distribution), the OTTPE broadcasts the set of parties present $\mathcal{P}_i$.*

In other words, for every event, all parties participating know which parties participate, and which parties arrived or left (implicitly). However, it is private who provides input, except newly arrived parties, as those always have to provide input.

Protocol 3 shows how parties can determine that an event happens without a party broadcasting that it wants to provide input.

Protocol 3 (Emulating an OTTPE[41]). *The overall protocol is similar to Protocol 2, and we only describe the main differences: The idea is to use a special offline functionality $h_i$ (and corresponding protocol $\varpi_i$) that will determine whether an event happens, i. e., at least one party wants to provide input. Each party currently present provides its actual input if it wants to provide input, and otherwise they send $\perp$. The special offline functionality $h_i$ outputs "1" if the input of at least one party is not $\perp$, and otherwise "0".*

*The evaluation of the decision functionality is now as follows: All parties jointly execute the special offline protocol $\varpi_i$. If the output is "0", then compute $\varpi_i$ again. If the output is "1", then the parties continue executing $\pi_i$ (as done in Protocol 1) with the inputs used in $\varpi_i$. Note, the "inputs" $\perp$ are ignored for the evaluation of $\pi_i$. After the outputs are distributed, the parties can essentially continue with executing $\varpi_i$ again.*

*If at least one party arrives, then they execute $\varpi'_i$, which is a slightly modified version of $\varpi_i$: Protocol $\varpi'_i$ essentially behaves like $\varpi_i$, but it additionally does not accept $\perp$ as input from parties that just arrived, i. e., it enforces that a party that just arrived will always have to provide actual input, e. g., assuming the approach that $\perp$ is encoded with an additional bit (cf. Section 5), then this bit is just not part of the input for parties that just arrived. Therefore, $\varpi'_i$ will always evaluate to 1 and thus cause the evaluation of $\pi_i$. This is required as a party that arrives will provide input (in the ideal model), and thus the OTTPE will evaluate the decision function, i. e., arriving parties cause an event to happen. If $\varpi_i$ would be executed unchanged, and a new (malicious) party provides $\perp$ as input to $\varpi_i$, then this would prevent the evaluation of the decision functionality, and thus differ from the OTTPE.*

*Regarding the initial ramp-up phase, we must ensure that only the input of parties that arrived or are present in an event $\sigma_i$ are allowed to actually provide input to $\pi_i$. Note, the events are implicitly defined by the parties arriving in certain time windows. However, this can easily be achieved by essentially using $\varpi'_i$ again, but now $\varpi'_i$ additionally accepts only inputs from the corresponding parties that are already present or just arrived for the corresponding events (time windows).*

*Finally, note that as multiple parties can arrive in the same event, events can happen only at certain points in time, e. g., each 5 minutes, in order to allow parties to arrive (parties cannot arrive at the exact same time). This implies, that at certain points in time an event could have occurred. Therefore, whenever such a point in time is passed, then $\varpi_i$ (or $\varpi'_i$) needs to be executed. After the ramp-up phase, if between two "possible" events, the number of parties present does not change, then this is already covered by repeatedly executing $\varpi_i$ as described above. Then, if $\varpi_i$ outputs "1" an event has happened, and if it outputs "0" then no event happened. For the ramp-up phase this has to be done, too: Events where parties arrive, are already covered above. And for "possible" events that could have occurred without parties arriving we have to execute $\varpi'_i$ accordingly, in order to evaluate the corresponding events in retrospect.*

Theorem 4. *Protocol 3 perfectly emulates an OTTPE for any online functionality with decision functionality $\overset{\infty}{f}$ in the presence of an adversary $A_{am}$ and a creator $C_{<\frac{1}{3}}$ with $\tau := 3$, or in the presence of an adversary $A_{sh}$ and a creator $C_{<\frac{1}{2}}$ with $\tau := 2$. This holds only if $n \in \mathbb{N}$ and $n > \tau$, and initially at least n parties arrive first without providing input (while their arrival is broadcast by the OTTPE), and then those n parties provide their inputs, and then at least n parties are always present.*[42]

[38]Correction: In the original version [24] parties just arrived in an event. However, as "arriving" and "providing input" is now separated, we included a clarification of how multiple parties can arrive at the same time.

[39]Correction: In the original version [24] parties just arrived in an event. However, as "arriving" and "providing input" is now separated, we included an approach of how multiple parties can arrive at the same time and how multiple parties can provide input.

[40]Correction: In the original version [24] the OTTPE broadcasts the parties for each event. However, as this is now split in having parties first "arrive" and then "provide input", the OTTPE also needs to first broadcast the parties that arrived. In addition, we modeled the behavior more explicitly.

[41]Extension: We added more details of the protocol, especially w. r. t. the ramp-up phase.

[42]Correction: In the original version [24] we just stated that "initially at least *n* parties arrive", but the intended meaning was unclear. This is now corrected by specifying this more precisely.

Proof. The proof is similar to the one of Theorem 2. According to Theorem 1 there exists a protocol $\varpi_i$ that perfectly emulates $h_i$ (just repeatedly simulate the execution of $h_i$ until a party wants to actually provide input). The same holds for $\varpi'_i$ and the ramp-up phase. Note, if the OTTPI broadcasts that an event happens, then the simulator needs to ensure that $\varpi_i$ evaluates to "1", which however the simulator can easily do. Hence, also the combined execution of $\varpi_i$ (or $\varpi'_i$) and $\pi_i$ perfectly emulates the combined execution of $h_i$ and $f_i$. The existence of a simulator $S$ then follows directly, and $S$ behaves similar as in the proof of Theorem 2. □

If each party is allowed to provide input in multiple events, then parties can arrive early without providing actual input, e. g., a party can arrive and participate in the evaluation by first only sending a "hello", and later its actual input. Similar, a party can leave early, e. g., sending a "bye" without being matched at all. In addition, a party can leave late, if the removal from the state is triggered by the party itself, e. g., disregarding whether a party is matched or not, the party is only removed from the state if the party sends a "bye".

However, allowing such functionalities makes it desirable to hide when an event happens: Consider the RANDOM matching algorithm with $n$ servers (cf. Section 6). Assume that at least $m \geq n$ parties arrived and no party left. After at most $m + n$ events ("hello" and actual input) we know that at least $n$ of the $m$ parties are matched. If $m = n$ then we know which parties are matched. However, if we cannot count the events w. r. t. providing actual input, then we cannot deduce that. Therefore, we introduce again a new OTTP version that partially hides when an event happens.

Definition 11 (OTTP with public Participation[43]). *An* OTTP with public participation (OTTPP) *is an ideal OTTP that additionally behaves as follows:*

- *Whenever one or multiple parties arrive, the OTTPP broadcasts the parties that arrive before any party can provide input.*
- *Whenever after a decision $\gamma_i$ (and possible output distribution) the set of parties present changed ($\mathcal{P}_i \neq \mathcal{P}_{i-1}$), the OTTPP broadcasts the set of parties present $\mathcal{P}_i$. The index $i$ is not known to any party.*

Protocol 4 (Emulating an OTTPP[44]). *The overall protocol is similar to Protocol 3, and we only describe the main differences: Instead of using a special offline functionality $h_i$ (protocol $\varpi_i$) to determine that an event happens, we will combine $h_i$ and $f_i$ into $f'_i$ (combine $\varpi_i$ and $\pi_i$ into $\pi'_i$) and repeatedly evaluate $f'_i$, i. e., $\pi'_i$ will not output "1" or "0" depending on whether an actual input was provided or not: Parties that want to provide input use their actual input and their share of the state $\phi_i$ as input for $\pi'_i$. Parties without actual input use $\perp$ and their share of the state $\phi_i$ as input for $\pi'_i$. If all parties send $\perp$, then $\pi'_i$ (i. e., $f'_i$) computes a dummy evaluation of $\overset{\infty}{f}$ and outputs $(\phi_i, \emptyset)$. If at least one party sends an actual input, then $\pi'_i$ (i. e., $f'_i$) will evaluate $\overset{\infty}{f}$ with the actual inputs and state, and distribute the outputs as done before by $\pi_i$ (i. e., $f_i$). After the outputs are distributed, the parties can essentially continue with executing $\pi'_i$ again.*

*Similar as in Protocol 3, the evaluation of the decision functionality has to be done whenever new parties arrive, also the ramp-up phase has to consider which parties are actually present in each event, and "possible" events (caused by the time-window for each event) have to be considered accordingly. This includes especially events in the ramp-up phase where parties provide input multiple times, without new parties arriving in between. These situations can be handled analogously as already described in Protocol 3, and are thus not repeated.*

Theorem 5. *Protocol 4 perfectly emulates an OTTPP for any online functionality with decision functionality $\overset{\infty}{f}$ in the presence of an adversary $A_{am}$ and a creator $C_{<\frac{1}{3}}$ with $\tau := 3$, or in the presence of an adversary $A_{sh}$ and a creator $C_{<\frac{1}{2}}$ with $\tau := 2$. This holds only if $n \in \mathbb{N}$ and $n > \tau$, and initially at least $n$ parties arrive first without providing input (while their arrival is broadcast by the OTTPP), and then those $n$ parties provide their inputs, and then at least $n$ parties are always present.*[45]

Proof. The proof is similar to the ones of Theorems 2 and 4. According to Theorem 1 there exists a protocol $\pi'_i$ that perfectly emulates the offline functionality $f'_i$. The existence of a simulator $S$ then follows directly, and $S$ behaves similar as in the proof of Theorem 2. □

Remark 5 (Partial privacy with public arrival). *The greedy algorithm for online matching with general arrival can guarantee* partial *privacy with public arrival if we allow parties to stay although they are already matched and allow them to leave although they are not yet matched. If no party arrives or leaves, then no event happens. If at least one party arrives, and no party leaves, then either no party is matched, or at least one party is matched and decided to stay. If at least one party arrives, and at least one party leaves, then either no party is matched and at least one party decided to leave unmatched, or at least one party is matched and left. If no party arrives, and at least one party leaves, then the party leaving is either matched and waited till now, or left unmatched. In all cases, nothing can be learned. However, it is not perfect: Assume that initially five parties $p_1, \ldots, p_5$ arrive, where $p_4$ and $p_5$ are controlled by a semi-honest adversary. Assume the matches are $(p_2, p_4)$ and $(p_3, p_5)$, but no party leaves. Now, assume $p_1$ and $p_5$ leave (preserve honest majority), and the evaluation continues with other parties arriving. If $p_1$ never arrives again, then the adversary learns that $p_1$ leaves unmatched.*[46]

W. r. t. the amount of information leaked, it holds that OTTPI $\geq$ OTTPE $\geq$ OTTPP $\geq$ ideal OTTP (equality holds only for certain functionalities): The ideal OTTP leaks nothing which is not specified by the online functionality. If a party arrives or leaves, the OTTPI, the OTTPE and the OTTPP leak that an event happens. Events where no parties arrive or leave stay hidden in the OTTPP model, but are known by every participating party in the OTTPE and the OTTPI model. Note, parties providing input or obtaining

[43]Correction: In the original version [24] the OTTPP broadcasts the set of parties only when it changes. We here make it more explicit that arriving parties are always broadcast, before any party can provide input, too. In addition, we modeled the behavior more explicitly.

[44]Extension: We added some additional information for clarification.

[45]Correction: In the original version [24] we just stated that "initially at least $n$ parties arrive", but the intended meaning was unclear. This is now corrected by specifying this more precisely.

[46]Correction: In the original version [24] $p_1$ and $p_2$ where corrupted, which is strictly speaking not possible assuming the creator ensures an honest majority at all times. Therefore, we changed the corrupted parties to be $p_4$ and $p_5$ to be consistent.

output will always learn that an event happened. Only the OTTPI leaks always (for every event) which parties provide input.

## C.3 A Note on Efficiency and Practicality

Protocols 2, 3 and 4 emulating the OTTPI, OTTPE and OTTPP imply very different impacts on computational efficiency. They indicate that the amount of information we want to hide correlates with the computation overhead. Protocol 4 emulating an OTTPP is by far the most inefficient, as the actual computation has to be done continuously. Protocol 3 emulating an OTTPE is more efficient, as the special offline protocol $\varpi_i$ (or $\varpi'_i$) is rather simple and can easily be called frequently. Protocol 2 emulating an OTTPI is obviously the most efficient one, as computations are only performed when actually required.

As discussed in Appendix B, the arrival of parties seems rather hard to hide in a purely distributed online setting. Thus, for now the OTTPI, OTTPE and OTTPP model seem to be the only reasonable (and possibly feasible) models to study.

## D CLASSICAL SMPC

We use the description of Goldreich [19] of SFE and reactive SMPC as basis for Definitions 12 to 15 and the corresponding TTPs in Definitions 16 and 17. However, we adapt it such that it is consistent and easier comparable with our definition of online functionalities in Section 4.[47]

### D.1 Offline Functionalities

An offline functionality is a function mapping inputs to outputs.

Definition 12 (Offline Functionality). *Let $\mathcal{X}$ be an input space and let $\mathcal{Y}$ be an output space. An* offline functionality *$f : \mathcal{X}^n \to \mathcal{Y}^n$ is defined as $(y_1, \dots, y_n) := f(x_1, \dots, x_n)$ with $x_\ell \in \mathcal{X}$ and $y_\ell \in \mathcal{Y}$ for $\ell \in \{1, \dots, n\}$.*

A simple offline functionality is the function $f(x_1, x_2) := (x_1 + x_2, x_1 + x_2)$ which computes the sum of two inputs.

The evaluation of an offline functionality with the private inputs of parties is defined as follows:

Definition 13 (Evaluation of an Offline Functionality). *Let $\mathcal{P} := \{p_1, \dots, p_n\}$ be a set of parties and let $x_1, \dots, x_n \in \mathcal{X}$ be their corresponding private inputs.*

*An* offline functionality *$f : \mathcal{X}^n \to \mathcal{Y}^n$ is* evaluated *by computing $(y_1, \dots, y_n) := f(x_1, \dots, x_n)$. Party $p_\ell \in \mathcal{P}$ will obtain output $y_\ell \in \mathcal{Y}$ for $\ell \in \{1, \dots, n\}$.*

### D.2 Reactive Functionalities

A reactive functionality maps iteratively inputs to outputs given a state. In one iteration the reactive function computes the outputs and updates the state.

Definition 14 (Reactive Functionality). *Let $\mathcal{X}$ be an input space, let $\mathcal{Y}$ be an output space and let $\Phi$ be a state space. A* reactive functionality *maps an ordered sequence of inputs $(x_{1,1}, \dots, x_{1,n})$, $(x_{2,1}, \dots, x_{2,n}), \dots \in \mathcal{X}^n$ to an ordered sequence of outputs $(y_{1,1}, \dots, y_{1,n}), (y_{2,1}, \dots, y_{2,n}), \dots \in \mathcal{Y}^n$ by iteratively computing a* reactive function *$\overline{f} : \Phi \times \mathcal{X}^n \to \Phi \times \mathcal{Y}^n$. The reactive function is defined as $(\phi_i, y_{i,1}, \dots, y_{i,n}) := \overline{f}(\phi_{i-1}, x_{i,1}, \dots, x_{i,n})$ where $\phi_0 \in \Phi$ is the initial state.*

A simple reactive functionality is the function $\overline{f}(\phi_{i-1}, x_{i,1}, x_{i,2}) := (\phi_{i-1} + x_{i,1} + x_{i,2}, \phi_{i-1}, \phi_{i-1})$ with $\phi_0 := 0$ which computes the sum of all previous inputs over time.

The evaluation of a reactive functionality with the private inputs of parties is defined as follows:

Definition 15 (Evaluation of a Reactive Functionality). *Let $\mathcal{P} := \{p_1, \dots, p_n\}$ be a set of parties and let $(x_{i,1}, \dots, x_{i,n}) \in \mathcal{X}^n$ be their corresponding i-th private inputs. Let $\phi_0 \in \Phi$ be the initial state.*

*For each input $(x_{i,1}, \dots, x_{i,n})$ the reactive function $\overline{f} : \Phi \times \mathcal{X}^n \to \Phi \times \mathcal{Y}^n$ is* evaluated *as $(\phi_i, y_{i,1}, \dots, y_{i,n}) := \overline{f}(\phi_{i-1}, x_{i,1}, \dots, x_{i,n})$. After each evaluation, party $p_\ell \in \mathcal{P}$ will obtain output $y_{i,\ell} \in \mathcal{Y}$ for $\ell \in \{1, \dots, n\}$.*

### D.3 Offline Trusted Third Party

Next we define an *offline TTP* that allows parties to securely evaluate an offline functionality. An offline TTP receives the inputs of the participating parties, evaluates the functionality, and finally distributes to each participating party is prescribed output. After this process, the evaluation is finished.

Definition 16 (Offline TTP). *An* offline TTP *evaluates an offline functionality $f$ for a set of parties $\mathcal{P} := \{p_1, \dots, p_n\}$ as follows:*

1. *All parties send their private inputs $x_1, \dots, x_n$ to the offline TTP.*
2. *The offline TTP computes $(y_1, \dots, y_n) := f(x_1, \dots, x_n)$.*
3. *The offline TTP sends the output $y_j$ to party $p_j$ for $j \in \{1, \dots, n\}$.*

An offline TTP cannot evaluate an online functionality, as it neither allows the parties to provide input over time nor is capable of storing a state.

### D.4 Reactive Trusted Third Party

In contrast to an offline TTP that is finished after providing outputs, a *reactive TTP* continuously receives inputs and provides outputs.

Definition 17 (Reactive TTP). *Let the parties $\mathcal{P} := \{p_1, \dots, p_n\}$ provide their inputs $(x_{1,1}, \dots, x_{1,n}), (x_{2,1}, \dots, x_{2,n}), \dots$ in an ordered sequence. An* reactive TTP *evaluates a reactive functionality with reactive function $\overline{f} : \Phi \times \mathcal{X}^n \to \Phi \times \mathcal{Y}^n$ as follows: The reactive TTP stores the initial state $\phi_0$.*

1. *The reactive TTP receives all i-th private inputs $x_{i,1}, \dots, x_{i,n}$.*
2. *The reactive TTP computes $(\phi_i, y_{i,1}, \dots, y_{i,n}) := \overline{f}(\phi_{i-1}, x_{i,1}, \dots, x_{i,n})$.*
3. *The reactive TTP sends the i-th output $y_{i,j}$ to party $p_j$ for $j \in \{1, \dots, n\}$.*
4. *The reactive TTP stores the new state $\phi_i$ and deletes the old state $\phi_{i-1}$.*

A reactive TTP cannot evaluate an online functionality, as it requires a fixed set of parties that is known a priori, and it does not allow to change the number of parties participating over time.

[47] Correction: In the original version [24] parties obtained output only if it was not $\perp$. However, as here now $\perp \notin \mathcal{X}$ and $\perp \notin \mathcal{Y}$, this condition is now removed in the corresponding definitions here in Appendix D, too.