Legally Enforceable Fairness in Secure Two-Party Computation[*]

Yehuda Lindell[†]

Department of Computer Science
Bar-Ilan University, ISRAEL.
lindell@cs.biu.ac.il

Abstract

In the setting of secure multiparty computation, a set of mutually distrustful parties wish to securely compute some joint function of their private inputs. The computation should be carried out in a secure way, meaning that the properties *privacy*, *correctness*, *independence of inputs*, *fairness* and *guaranteed output delivery* should all be preserved. Unfortunately, in the case of no honest majority – and specifically in the important two-party case – it is impossible to achieve fairness and guaranteed output delivery. In this paper, we show how a legal infrastructure that respects digital signatures can be used to enforce fairness in two-party computation. Our protocol has the property that if one party obtains output while the other does not (meaning that fairness is breached), then the party not obtaining output has a digitally signed cheque from the other party. Thus, fairness can be "enforced" in the sense that any breach results in a loss of money by the adversarial party.

1 Introduction

In the setting of secure multiparty computation, a set of parties with private inputs wish to jointly compute some functionality of their inputs. Loosely speaking, the security requirements of such a computation are that nothing is learned from the protocol other than the output (privacy), that the output is distributed according to the prescribed functionality (correctness), that parties cannot make their inputs depend on other parties' inputs (independence of inputs), that the adversary cannot prevent the honest parties from successfully computing the functionality (guaranteed output delivery), and that if one party receives output then so do all (fairness). The generality of secure multiparty computation has made it a very important and useful tool for proving the feasibility of carrying out a variety of tasks. Indeed, in a number of different settings, it has been shown that *any* two-party or multiparty function can be securely computed [21, 14, 13, 3, 8]. Stated more simply, any distributed task that a set of parties wish to compute, can be computed in a secure way. This implies feasibility for a multitude of tasks, including those as simple as coin-tossing and agreement, and as complex as electronic voting, electronic auctions, electronic cash schemes, anonymous transactions, and privacy-preserving data mining.

[*]An extended abstract of this work appeared at *CT-RSA 2008*.

[†]Some of this work was carried out for Aladdin Knowledge Systems.

Fairness. Unfortunately, the above description is misleading and inaccurate. Indeed, it is possible to securely compute any functionality, where security implies all of the properties described above. However, this is only true if there exists an *honest majority* amongst the participating parties. In the case of no honest majority, and specifically in the important two-party case, it is *impossible* to achieve fairness and guaranteed output delivery [10] in general (although some non-trivial functions can be securely computed; see [12]). A number of different approaches have been taken to achieve some sort of fairness despite this impossibility:

1. *Gradual release:* In this approach, the output is not revealed all at once. Rather, it is released gradually with the property that if an abort occurs, then the adversary has not learned much more about the output than the honest parties. The drawback of this approach is that it is inherently expensive (requiring many rounds), and if the adversary is more powerful than the honest parties, fairness may still be breached. See, for example, the early works of [2, 15] and more recent works [20, 11] (a good survey of the many works in this area can be found in [20]).

2. *The optimistic model:* In this approach, a trusted server is used with the following property. If all the parties behave honestly, then there is no need for the server (it is not contacted). However, if fairness is breached, then the server may be contacted in which case fairness is restored. This approach can be highly efficient. However, its drawbacks are the need for new infrastructure (in the form of such a server) and the fact that the server must be *trusted* to not collude with the adversary. See [18, 19, 1, 4] for some works in this area.

In this paper, we present an approach to achieving fairness, that is inspired by the work of [9] and has some similarities with the optimistic model. The authors of [9] consider the question of fair exchange of signatures. They make a highly interesting observation that in order for a signature to be *enforced*, it needs to be presented at a court of law. In this case, the other party will actually see the signature. They therefore run a computation with the following property: the first party to receive output obtains something called a "keystone", and the second party then receives its signature (i.e., the signature it should receive as output). The keystone by itself gives nothing and so if the first party aborts after receiving it, no damage has been done (fairness has not been breached). In contrast, after receiving its signature, the second party may abort and the first party is left only with a useless keystone. Nevertheless, the interesting property here is that given the keystone *and* the second party's signature (i.e., the signature that the second party received as output), it is possible to construct the signature that the first party should receive as output. Thus, if the second party wishes to enforce its signed contract in a court of law, it can only do so by essentially revealing the signature that the first party should receive, thereby restoring fairness. In [9], the above notion is formalized and called a concurrent signature scheme. In addition, [9] present an efficient construction of such a scheme that is secure under the discrete logarithm assumption in the random oracle model.

Our results. In this paper, we extend the idea of [9] to general secure two-party computation. One byproduct is that we show that the problem of concurrent signatures can easily be cast as a standard secure two-party computation problem, and therefore random oracles are not necessary. We stress, however, that in contrast to the construction of [9], our protocol is *not* efficient. Thus, it should be viewed as a "feasibility proof" that concurrent signatures can be constructed under general assumptions and without random oracles.

The basic idea of our approach is as follows. We construct a protocol with the property that either both parties receive output (and so fairness is preserved) or one party receives output while the other receives a digitally-signed cheque from the other party that it can take to a court of law or a bank. This cheque can contain any sum of money, as agreed by the parties. The protocol further has the property that the only way that a party can evade paying the sum in the cheque is to reveal the other party's output, thereby restoring fairness.

It is instructive to compare our approach to the *optimistic model*. On the one hand, both solutions use a trusted party that is only contacted in the case of attempted cheating. However, the optimistic model guarantees fairness always, whereas our approach allows an adversary to breach fairness as long as it is willing to pay the cheque (although if the sum in the cheque is set appropriately, such an event is unlikely to ever occur). In addition, even if the adversary is not willing to pay the cheque, it can prevent the other party from receiving its output until the court or bank processes the cheque, at which point it can provide the other party with its output and evade payment. Thus, our approach provides a somewhat weaker security guarantee. On the other hand, in the optimistic model a dedicated server must be set up and trusted. The advantage of our model is that it uses existing infrastructure (like courts and banks) that *are* trusted. However, our approach does assume that digital signature law and digital cheques are respected, and so may not always be applicable. In summary, we believe that our approach provides an interesting alternative to the optimistic one.

Organization. As a warm-up to see how our construction works, we first present a simple solution to the problem of concurrent signatures in Section 2 that is based on general protocols for secure computation. Then, in Section 3 we present the definitions that we need, as well as a formal definition of *legally enforceable fairness*. Finally, in Section 4 we show how every two-party functionality can be securely computed with legally enforceable fairness.

2 Concurrent Signatures

As a warm-up, we present a protocol for concurrent signatures that is based on general secure two-party computation. We rely on the intuitive description in the Introduction of what concurrent signatures are. A formal definition can be found in [9]. We also use a general protocol for secure computation, where party P_1 always receives output first (e.g., the protocols of [14, 13] have this property); see definitions in Section 3 below.

We assume a public-key infrastructure for digital signatures. In particular, P_1 has a pair of signing/verification keys denoted (sk_1, vk_1) and P_2 has an analogous pair (sk_2, vk_2). Furthermore, each party knows the other's public verification key. Without loss of generality, we assume that a signature includes the message being signed upon. The aim of the protocol is for P_1 to receive a message m_2 that is signed by P_2, and for P_2 to receive a message m_1 that is signed by P_1. The protocol appears below in Figure 1.

We now informally describe why our protocol achieves concurrent signatures. First, observe that if both parties received output in the secure protocol, then they both have mutual signatures on the appropriate messages m_1 and m_2. However, if only P_1 receives output (because it receives output first), then P_2 does not receive the signature it should receive on the message m_1. Nevertheless, the signature σ_2 obtained by P_1 *contains* the signature σ_1 that P_2 should receive. Therefore, if P_1 wishes to enforce its signature by taking P_2 to court, it will necessarily reveal σ_1 – the signature that P_2 should receive – thereby restoring fairness. Relying on the constructions of secure protocols by [14, 13], for example, we have the following theorem:

Concurrent Signatures

The parties use a secure two-party protocol to compute the following functionality:

- **Inputs:**
  1. Party P_1 inputs its pair of keys (sk_1, vk_1), party P_2's verification-key vk_2, and the messages m_1 and m_2 to be signed upon
  2. Party P_2 inputs its pair of keys (sk_2, vk_2), party P_1's verification-key vk_1, and the messages m_1' and m_2' to be signed upon
- **Outputs:**
  1. If the keys do not match (i.e., P_2 inputs vk_1' that is different to the vk_1 input by P_1 or vice versa), or they are not valid (i.e., sk_1 is not the signing key associated with vk_1),[1] or $(m_1, m_2) \neq (m_1', m_2')$, then the functionality outputs $\perp$ to both parties. Otherwise:
  2. Party P_1 receives $\sigma_2 = \mathsf{Sign}_{sk_2}(m_2, \sigma_1)$, where σ_1 is defined next.
  3. Party P_2 receives $\sigma_1 = \mathsf{Sign}_{sk_1}(m_1)$; recall that by our convention a signature contains the message and so σ_1 contains m_1.

Figure 1: A protocol for concurrent signatures without random oracles

Theorem 1 *Assuming the existence of enhanced trapdoor permutations, there exist protocols for concurrent signatures, as defined in* [9].

We remark that the constructions of [9] rely on specific assumptions and assume a random oracle, whereas our construction relies only on general assumptions and is in the standard model. However, the constructions of [9] are highly efficient, while ours are not.

We also remark that P_1 does not receive a "pure" signature on m_2. Rather, it receives a signature on m_2 together with σ_1. This may have ramifications in some applications. For example, consider the case that P_1 wishes to show a third party a signature on m_2 without revealing σ_1 or m_1 (since σ_2 contains σ_1 which in turn contains m_1, this information is revealed). If necessary, it is possible to prevent this by defining that P_1 receives a signature on m_2 together with an *encryption* of σ_1 under a key belonging to P_2. This will have the same effect regarding fairness, but now σ_2 does not reveal anything about σ_1 or m_1 to a third party, as desired.

3 Definitions

3.1 Standard Definitions

We use the standard definition of two-party computation for the case of no honest majority, where no fairness is guaranteed. In particular, this means that the adversary always receives output first, and can then decide if the honest party also receives output. We remark that the definition in [13, Section 7] only allows a corrupted P_1 to receive output without the honest party receiving output. The variant that we use is obtained via a straightforward modification to the ideal model; see [16]. We refer the reader to [13, Section 7] for full definitions of security for secure two-party computation, and present a very brief description here only.

[1] We assume that such validity can be verified given the signing and verification key-pair. This is without loss of generality.

Preliminaries. A function $\mu(\cdot)$ is negligible in n, or just negligible, if for every positive polynomial $p(\cdot)$ and all sufficiently large n's it holds that $\mu(n) < 1/p(n)$. A probability ensemble $X = \{X(a,n)\}_{a\in\{0,1\}^*;n\in\mathbb{N}}$ is an infinite sequence of random variables indexed by a and $n \in \mathbb{N}$. (The value a will represent the parties' inputs and n the security parameter.) Two distribution ensembles $X = \{X(a,n)\}_{n\in\mathbb{N}}$ and $Y = \{Y(a,n)\}_{n\in\mathbb{N}}$ are said to be computationally indistinguishable, denoted $X \overset{c}{\equiv} Y$, if for every non-uniform polynomial-time algorithm D there exists a negligible function $\mu(\cdot)$ such that for every $a \in \{0,1\}^*$,

$$|\Pr[D(X(a,n)) = 1] - \Pr[D(Y(a,n)) = 1]| \leq \mu(n)$$

All parties are assumed to run in time that is polynomial in the security parameter. (Formally, each party has a security parameter tape upon which the value 1^n is written. Then the party is polynomial in the input on this tape.)

Secure two-party computation. A two-party protocol problem is cast by specifying a randomized process that maps sets of inputs to sets of outputs (one for each party). This process is called a functionality and is denoted $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, where party P_1 is supposed to receive the first output and party P_2 the second output. For simplicity, we assume that $f_1 = f_2$ and thus that both parties receive the same output. When considering protocols for securely computing any functionality, this is without loss of generality (encryption can be used so that P_1 cannot read P_2's portion of the output and vice versa).

Security is formalized by comparing a real protocol execution to an ideal model setting where a trusted party is used to carry out the computation. In this ideal model, the parties send their inputs to the trusted party who first sends the output to the adversary. (The adversary controls one of the parties and can instruct it to behave arbitrarily) After the adversary receives the output it either sends continue to the trusted party instructing it to also send the output to the honest party, or halt in which case the trusted party sends $\perp$ to the honest party. The honest party outputs whatever it received from the trusted party and the adversary outputs whatever it wishes. We stress that the communication between the parties and the trusted party is ideally secure. The pair of outputs of the honest party and an adversary $\mathcal{A}$ in an ideal execution where the trusted party computes f is denoted $\text{IDEAL}_{f,\mathcal{A}(w)}(x_1, x_2, n)$, where x_1, x_2 are the respective inputs of P_1 and P_2, w is an auxiliary input received by $\mathcal{A}$, and n is the security parameter.

In contrast, in the real model, a real protocol π is run between the parties without any trusted help. Once again, an adversary $\mathcal{A}$ controls one of the parties and can instruct it to behave arbitrarily. At the end of the execution, the honest party outputs the output specified by the protocol π and the adversary outputs whatever it wishes. The pair of outputs of the honest party and an adversary $\mathcal{A}$ in an real execution of a protocol π is denoted $\text{REAL}_{\pi,\mathcal{A}(w)}(x_1, x_2, n)$, where x_1, x_2, w and n are as above.

Finally, we present the notion of a "hybrid model" where the parties run a protocol π as well as having access to a trusted party. In this paper, we will use this to model external authorities that exist in the real world. For example, we will consider a certificate authority (for a public-key infrastructure) and a bank. In this model, the protocol π contains both standard messages that are sent between the parties as well as ideal messages that are sent between the parties and the trusted party. The pair of outputs of the honest party and an adversary $\mathcal{A}$ in a hybrid execution of a protocol π with a trusted party computing a functionality g is denoted $\text{HYBRID}^{g}_{\pi,\mathcal{A}(w)}(x_1, x_2, n)$, where x_1, x_2, w and n are as above.

Given the above, we can now define the security of a protocol π (we present this for the hybrid model because our protocols are in this model and by taking g to be a function with no output we have the real model as well).

Definition 2 *Let π be a probabilistic polynomial-time protocol and let f be a probabilistic polynomial-time two-party functionality. We say that π* securely computes f with abort in the g-hybrid model *if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ attacking π there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model so that for every $x_1, x_2, w \in \{0,1\}^*$,*

$$\left\{\text{IDEAL}_{f,\mathcal{S}(w)}(x_1, x_2, n)\right\}_{n\in\mathbb{N}} \stackrel{c}{\equiv} \left\{\text{HYBRID}^{g}_{\pi,\mathcal{A}(w)}(x_1, x_2, n)\right\}_{n\in\mathbb{N}}$$

This is called "security with abort" because when fairness is not guaranteed, the adversary is allowed to abort early (after it received output but before the honest party receives output).

Immediate message receipt. Any protocol that aims to achieve any notion of fairness must deal with the question of when to declare that one of the parties has *not* sent a message. This can be dealt with by introducing time into the model and allowing only a certain delay (as would be the case in practice), or can be achieved by assuming synchronous communication (i.e., the protocol proceeds in rounds and in each round all parties send and receive messages). For the sake of simplicity, we assume the latter. Furthermore, we assume that the receipt of messages is *immediate*, meaning that after a party receives a message, its next-message is sent straight away. In particular, this means that if it does not send such a message, it will not send it later.

Secure signature schemes. A signature scheme consists of three probabilistic polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ such that for every n, every (vk, sk) in the range of $\mathsf{Gen}(1^n)$, and every $m \in \{0,1\}^*$, $\mathsf{Vrfy}(vk, m, \mathsf{Sign}(sk, m)) = 1$. A signature scheme is existentially secure against chosen message attacks if any non-uniform probabilistic polynomial-time adversary given access to a signing oracle can generate a forgery with at most negligible probability. (A forgery is a valid signature on *any* message that the adversary did not query to its signing oracle.) See [13] for full definitions. By convention, we assume that a signature $\sigma = \mathsf{Sign}(sk, m)$ *contains the message m*. Thus, in order to verify a signature, it suffices to have the verification key vk and σ and so we can write $\mathsf{Vrfy}(vk, \sigma)$ without m. We also assume that the validity of a signing pair can be verified given vk and sk (this is without loss of generality; in particular, the random coins used to run Gen can be included as part of sk).

The certificate-authority functionality. We assume a public-key infrastructure for our protocols, and formalize this via the certificate-authority functionality of [6]. This functionality provides simple register and retrieve instructions and is denoted $\mathcal{F}_{CA}$; see Figure 2.

Functionality $\mathcal{F}_{CA}$

1. Upon receiving the first message $(\mathsf{Register}, sid, v)$ from a party P, send $(\mathsf{Register}, sid, v)$ to the adversary. Upon receiving back ok from the adversary, check that $sid = P$ and that this is the first request from P. If yes, then record the pair (P, v); otherwise, ignore the message.
2. Upon receiving a message $(\mathsf{Retrieve}, sid)$ from a party P', send $(\mathsf{Retrieve}, sid, P')$ to the adversary, and wait for an ok from the adversary. Then, if there is a recorded pair (sid, v) output $(\mathsf{Retrieve}, sid, v)$ to P'. Else output $(\mathsf{Retrieve}, sid, \perp)$ to P'.

Figure 2: The ideal certification authority functionality $\mathcal{F}_{CA}$

3.2 A Simple Bank Functionality

Our notion of legally enforceable fairness assumes the existence of an external authority that can force parties to carry out some action. In today's society such an authority exists in the form of a court of law (or a bank that respects digital cheques), and we assume the existence of digital signature law that can be used to enforce payment when one party holds a *cheque* that has been digitally signed by another party. We use the following notation:

> **Cheques:** A cheque chq for $\$\alpha$ for a party P_j from a party P_i is a signed message of the form $\mathsf{cheque}(cid, P_i \to P_j, \alpha, z)$, where cid is a unique identifier and z is an auxiliary-information field (like a "notes" field on a regular paper cheque). A cheque is only valid when signed and so chq contains the information cid, $P_i \to P_j$, α and z in some standardized form, all signed with party P_i's signing key. Thus, denoting P_i's signing key-pair as (vk_i, sk_i), we have that a cheque is the signature $\mathsf{chq} = \mathsf{Sign}_{sk_i}(cid, P_i \to P_j, \alpha, z)$. Recall that by our convention, a signature also contains the message and so the signature is all that is needed.

We assume that a digitally signed message constitutes a legally binding cheque that is respected by banks and by courts of law.

We define a functionality that represents a "bank". It is not supposed to be a full-fledged abstraction of the banking system. Rather, it is a minimal functionality that is fulfilled by the real banking system in use today. The bank that we define is such that upon receiving a cheque $\mathsf{cheque}(cid, P_i \to P_j, \alpha, z)$, the bank transfers $\$\alpha$ from P_i's account to P_j's account, and sends P_i a copy of the cheque. (This is analogous to the scan of a cheque that is sent by many banks to customers.) For the sake of simplicity, we initialize the accounts of all parties to $0 and allow the current balance of an account to be any value, positive or negative (as such we do not check if a party has sufficient funds to cover a cheque payment). We denote the current balance of a party P_i by $\mathsf{balance}_i$. The bank functionality is defined as follows:

Functionality $\mathcal{F}_{\text{bank}}$

The functionality $\mathcal{F}_{\text{bank}}$ runs with a certificate authority $\mathcal{F}_{CA}$ and parties $P_1, \ldots, P_n$. It initializes values $\mathsf{balance}_i = 0$ for all i and a set $\mathsf{used} = \phi$, and works as follows:

- Before processing any messages, the functionality sends $(\mathsf{Retrieve}, i)$ to $\mathcal{F}_{CA}$, for every $i = 1, \ldots, n$.[2] If $\mathcal{F}_{\text{bank}}$ receives back any response with $\perp$ then it halts immediately. Otherwise, $\mathcal{F}_{\text{bank}}$ stores the keys $vk_1, \ldots, vk_n$ where vk_i denotes the key retrieved for party P_i, and proceeds to the next step.
- Upon receiving a message $\mathsf{chq} = \mathsf{cheque}(cid, P_i \to P_j, \alpha, z)$ from a party P_j, the functionality first checks that chq is a valid cheque from P_i to P_j (it does this using P_i's verification-key vk_i as retrieved from $\mathcal{F}_{CA}$), and that $(cid, i, j) \notin \mathsf{used}$. If not, it ignores the message. If yes, it does the following:
  1. Set $\mathsf{balance}_i = \mathsf{balance}_i - \alpha$ and $\mathsf{balance}_j = \mathsf{balance}_j + \alpha$
  2. Add (cid, i, j) to the set used; formally, $\mathsf{used} \leftarrow \mathsf{used} \cup \{(cid, i, j)\}$
  3. Send chq to party P_i

Figure 3: The ideal bank functionality $\mathcal{F}_{\text{bank}}$

[2] We assume that all parties have registered a public key with $\mathcal{F}_{CA}$ before $\mathcal{F}_{\text{bank}}$ begins. In particular, we assume that for every i, party P_i has sent $(\mathsf{Register}, i, vk_i)$ to $\mathcal{F}_{CA}$, where vk_i is the signature verification key of P_i.

We remark that $\mathcal{F}_{\text{bank}}$ records tuples of the form (cid, i, j) in the set used in order to ensure that the same cheque is not cashed twice. We also note that $\mathcal{F}_{\text{bank}}$ is actually parameterized by a digital signature scheme which is used for verification of the cheque signatures.

3.3 Legally Enforceable Fairness

We are now ready to formally define what it means for a protocol to securely compute a functionality f with legally enforceable fairness. The basic idea is that an adversary has three choices regarding the output of the protocol:

1. The adversary can abort the protocol before anyone learns anything (preserving fairness).
2. The protocol can conclude with both parties receiving output (preserving fairness).
3. The protocol can conclude with the adversary receiving output while the honest party does not. However, in case this happens, the honest party receives a cheque for \$$\alpha$ from the adversary that it can cash at the bank.

We formalize this by defining a functionality $\mathcal{F}_f^\alpha$ that incorporates the bank functionality and computes f as above; see Figure 4.

Functionality $\mathcal{F}_f^\alpha$

The functionality $\mathcal{F}_f^\alpha$ runs with parties P_1 and P_2, with initial balances $\mathsf{balance}_1$ and $\mathsf{balance}_2$, and an adversary $\mathcal{A}$. Let P_i denote the corrupted party and P_j the honest party ($i, j \in \{1, 2\}, i \neq j$). Functionality $\mathcal{F}_f^\alpha$ works as follows:

1. $\mathcal{F}_f^\alpha$ receives inputs x_1 and x_2 from parties P_1 and P_2. If either of the inputs equal $\perp$ or are invalid, then $\mathcal{F}_f^\alpha$ sends $\perp$ to both P_1 and P_2 and halts.
2. If $\mathcal{F}_f^\alpha$ receives two valid inputs, it sends $y = f(x_1, x_2)$ to $\mathcal{A}$ and waits for $\mathcal{A}$'s response.
   (a) If $\mathcal{A}$ replies with fair then $\mathcal{F}_f^\alpha$ sends y to P_j
   (b) If $\mathcal{A}$ replies with unfair (or doesn't reply), then $\mathcal{F}_f^\alpha$ sets $\mathsf{balance}_i = \mathsf{balance}_i - \alpha$ and $\mathsf{balance}_j = \mathsf{balance}_j + \alpha$.

Figure 4: The ideal functionality $\mathcal{F}_f^\alpha$ for computing f with legally enforceable fairness

We remark that by our definition, one party is always corrupted. This makes no difference because the case of two honest parties can be captured by considering an adversary that always replies with fair. This is then the same as a case where neither party is corrupted.

Definition 3 *Let π be a protocol and f a two-party functionality. We say that π* securely computes f with α-legally enforceable fairness *if π securely computes the functionality $\mathcal{F}_f^\alpha$ according to Definition 2.*

As a sanity check, we show that any protocol that securely computes f with legally enforceable fairness also securely computes f under the standard notion of security. In order to do this, however, we must make a small modification to the definition in the case of an unfair abort (i.e., where the adversary receives output but the honest party does not), so that the honest party does not necessarily output $\perp$ but may simply remain in a waiting state. (This is essentially equivalent, but the change is necessary on a technical level to make the models match.) Given this modification, we prove the following:

Claim 4 *Let π be a protocol that securely computes a two-party functionality f with α-legally enforceable fairness. Then, π securely computes f with abort, as in Definition* 2.

Proof: Let $\mathcal{A}$ be a real-world adversary that attacks protocol π. We wish to construct a simulator $\mathcal{S}$ for the standard notion of security as in Definition 2. In order to do this, we first consider the simulator $\mathcal{S}'$ that is guaranteed to exist by the fact that π securely computes f with legally enforceable fairness; note that $\mathcal{S}'$ is designed to work with $\mathcal{F}_f^\alpha$. We construct $\mathcal{S}$ from $\mathcal{S}'$ as follows:

1. When $\mathcal{S}'$ sends an input x to the trusted party, $\mathcal{S}$ forwards the same x to its trusted party. Then, when it receives back y, it hands it to $\mathcal{S}'$ and waits for $\mathcal{S}'$'s reply:

2. If $\mathcal{S}'$ replies with fair then $\mathcal{S}$ sends continue to the trusted party (indicating that the honest party should also receive output) and outputs whatever $\mathcal{S}'$ outputs.

3. If $\mathcal{S}'$ replies with unfair or does not reply, then $\mathcal{S}$ sends halt to the trusted party (indicating that the honest party should not receive output) and outputs whatever $\mathcal{S}'$ outputs.

It is clear that the joint output distribution of $\mathcal{S}$ and the honest party is identical to the joint output distribution of $\mathcal{S}'$ and the honest party (which in turn is indistinguishable from the output of $\mathcal{A}$ and the honest party in a real protocol execution). Therefore, the output distributions of $\mathcal{S}$ and $\mathcal{A}$ and the honest party are indistinguishable, as required. ■

4 A Protocol with Legally Enforceable Fairness

Before formally presenting the protocol, we informally describe how it works. The protocol has a number of phases, as follows:

1. **Registration phase:** In this phase, the parties register their digital signature keys with the certificate authority $\mathcal{F}_{CA}$.

2. **Main computation phase:** In this phase, the parties run a protocol for secure computation with the property that party P_1 receives a cheque for $\$\alpha$, while P_2 receives nothing. The important property of the cheque received by P_1 is that it itself contains an encrypted cheque for P_2 for $\$\alpha$, much like the signature-inside-a-signature in our solution for concurrent signatures in Section 2. We call this cheque for P_2 a counter-cheque because it enables P_2 to "counter" the cheque that P_1 has received. That is, if P_1 sends its cheque to the bank authority, then upon receiving a copy of this cheque from $\mathcal{F}_{\text{bank}}$, party P_2 can decrypt its own counter-cheque and return the balances to their original values. (Using this terminology, the cheque that P_1 receives contains an encrypted counter-cheque for P_2.) In addition to the above, and crucial to our solution as we will see below, the counter-cheque for P_2 contains the function output! We stress that the counter-cheque is *encrypted* and so only P_2 can obtain the output that is inside it.

3. **Output exchange phase:** In this phase, P_1 is supposed to send its cheque to P_2, upon which P_2 decrypts it, obtains the counter-cheque and sends it back to P_1. Following this, both parties read the function output from the counter-cheque and output it.

We now informally analyze the above flow in terms of what happens if one of the parties aborts before both have received output. We present our analysis in a step-by-step fashion, showing what happens if an abort occurs at any given step:

1. *Either party aborts in the main computation phase:* In this case no one learns anything and so fairness is preserved.

2. *P_1 aborts by not sending its cheque to P_2:* As in the previous case, here no one learns anything and so fairness is preserved. (Recall that P_1's output from the main computation phase is its cheque and this does not directly contain the output. Rather, the output only appears in the counter-cheque for P_2 that is encrypted so that only P_2 can decrypt it. This means that if P_1 does not send its cheque to P_2 then no one receives output.)

3. *P_2 aborts by not sending its counter-cheque back to P_1:* In this case, P_2 has already received output (because the output is contained in its counter-cheque), while P_1 has not. Thus, if P_2 aborts at this point, without sending P_1 the counter-cheque, fairness will have been breached. This is where the cheques and legal enforcement comes in. In this situation, P_1 has a cheque from P_2 for $\$\alpha$. Thus, the only way for P_2 to avoid paying $\$\alpha$ to P_1 is for it to present its counter-cheque. But, the counter-cheque contains the output and so if P_2 presents the cheque, fairness is restored!

We conclude that either fairness is preserved, or P_1 can force P_2 to pay it $\$\alpha$. We remark that now that cheques are included as part of the model, we must also ensure that it is not possible for a dishonest party to obtain a cheque that cannot be countered by an honest party. Otherwise an adversary could inflict financial damage on an honest party (note that this property is implicit in the definition of $\mathcal{F}_f^\alpha$ for legally enforceable fairness). Now, in our protocol informally described above, the cheque received by P_1 contains a counter-cheque for P_2. Therefore, if a corrupted P_1 sends this cheque to the bank, P_2 will receive a copy containing a counter-cheque that it can be used to restore the bank balances to their original values. Likewise, P_2 only receives its cheque after P_1 has received its own, and so P_1 can always counter any cheque sent by P_2 to the bank.

We are now ready to formally describe the protocol π_α for securely computing f with α-legally enforceable fairness. The protocol is in the $\mathcal{F}_{CA}, \mathcal{F}_{\text{bank}}$-hybrid model, meaning that the parties have access to a trusted certificate authority and a trusted bank (or court of law that enforces digital cheques), and appears below in Figure 5.

Theorem 5 *Assume that the protocol used in phase* 1 *is secure by Definition* 2 *and that the signature scheme is existentially unforgeable under chosen-message attacks. Then, Protocol π_α securely computes f with α-legally enforceable fairness in the $(\mathcal{F}_{CA}, \mathcal{F}_{\text{bank}})$-hybrid model.*

Proof: The proof is in the hybrid model, where we assume that $\mathcal{F}_{CA}$, $\mathcal{F}_{\text{bank}}$ and the computation of phase 1 are carried out with the help of a trusted third party (see [5] and [13]). We separately analyze the case that P_1 is corrupted and the case that P_2 is corrupted. Recall that what we need to show is that π_α securely computes the functionality $\mathcal{F}_f^\alpha$.

P_1 is corrupted. Let $\mathcal{A}$ be a (hybrid-model) adversary controlling party P_1. We construct a simulator $\mathcal{S}$ as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ upon its input x_1 and auxiliary-input α and n. In addition, $\mathcal{S}$ chooses a key-pair $(vk_2, sk_2) \leftarrow \mathsf{Gen}(1^n)$ for P_2.

2. When $\mathcal{A}$ sends a $(\mathsf{Retrieve}, P_2)$ message intended for $\mathcal{F}_{CA}$, simulator $\mathcal{S}$ replies with $(\mathsf{Retrieve}, P_2, vk_2)$.

3. When $\mathcal{A}$ sends a $(\mathsf{Register}, P_1, vk_1)$ message intended for $\mathcal{F}_{CA}$, simulator $\mathcal{S}$ records vk_1.

A Protocol π_α for Securely Computing f

Inputs: P_1 has x_1 and P_2 has x_2; both parties have α and n. **The protocol:**

1. **Phase 0 – registration:** Prior to any execution of the protocol:
   (a) P_1 chooses $(vk_1, sk_1) \leftarrow \mathsf{Gen}(1^n)$ and sends $(\mathsf{Register}, P_1, vk_1)$ to $\mathcal{F}_{CA}$.
   (b) P_2 chooses $(vk_2, sk_2) \leftarrow \mathsf{Gen}(1^n)$ and sends $(\mathsf{Register}, P_2, vk_2)$ to $\mathcal{F}_{CA}$.

2. **Phase 1 – main computation:** The parties use a secure two-party protocol to compute the following functionality:

   Inputs:

   (a) Party P_1 inputs its signing key-pair (vk_1, sk_1), party P_2's public key vk_2 (obtained by sending $(\mathsf{Retrieve}, P_2)$ to $\mathcal{F}_{CA}$), its input x_1, α and n. In addition, P_1 inputs a random string $cid_2 \in_R \{0,1\}^n$.
   (b) Party P_2 inputs its signing key-pair (vk_2, sk_2), party P_1's public key vk_1 (obtained by sending $(\mathsf{Retrieve}, P_1)$ to $\mathcal{F}_{CA}$), its inputs x_2, α and n. In addition, P_2 inputs a random string $cid_1 \in_R \{0,1\}^n$ and a random r of appropriate length (see below).

   Outputs: The functionality first checks that the α and n values received from P_1 and P_2 are the same, that the keys match (i.e., P_2 inputs $vk_1' = vk_1$ and vice versa), and that the key-pairs that are input are valid (i.e., sk_i is associated with vk_i).

   - If yes, then the functionality sets $cid = cid_1 \| cid_2$ and defines the outputs as follows:
     (a) Party P_1 receives the cheque $\mathsf{chq}_1 = \mathsf{cheque}(cid, P_2 \rightarrow P_1, \alpha, z)$, where $z = r \oplus \mathsf{chq}_2$, $\mathsf{chq}_2 = \mathsf{cheque}(cid, P_1 \rightarrow P_2, \alpha, y)$ and $y = f(x_1, x_2)$.
     (b) Party P_2 receives nothing.
   - If no, then both parties receive $\perp$, in which case they output $\perp$.

3. **Phase 2 – exchange outputs:**
   (a) If P_1 did not receive output from phase 1 it halts and outputs $\perp$. Otherwise, it sends chq_1 to P_2.
   (b) P_2 waits to receive a value chq_1 from P_1. Upon receiving such a chq_1, party P_2 checks that the identifier in chq_1 begins with cid_1 and that the cheque is valid with respect to vk_2. If not, P_2 ignores the message and continues waiting. Otherwise, P_2 computes $\mathsf{chq}_2 = r \oplus z$, sends chq_2 to P_1, and outputs the value y inside chq_2.
   (c) If P_1 did not receive chq_2 from P_2, or if the identifier in chq_2 does not end with cid_2, or if chq_2 is invalid with respect to vk_1, then P_1 sends chq_1 to $\mathcal{F}_{\text{bank}}$ (in order to receive \$$\alpha$). Else, P_1 outputs the value y inside chq_2.

4. **Additional instructions:**
   (a) If P_1 receives a valid chq_2 with an identifier ending with cid_2 from $\mathcal{F}_{\text{bank}}$ (indicating a payment made to P_2), then P_1 sends chq_1 with cid from the same execution to $\mathcal{F}_{\text{bank}}$.
   (b) If P_2 receives a valid cheque chq_1 with an identifier beginning with cid_1 from $\mathcal{F}_{\text{bank}}$ (indicating a payment made to P_1), then P_2 works as follows:
      i. If P_2 received already chq_1 from P_1, then it sends chq_2 with cid to $\mathcal{F}_{\text{bank}}$.
      ii. If P_2 did not receive chq_1 from P_1, it takes chq_1 as the value it is waiting for in Step 3b above and proceeds according to those instructions. In addition it sends chq_2 to $\mathcal{F}_{\text{bank}}$, as derived in Step 3b.

Figure 5: A protocol for computing f with α-legally enforceable fairness

4. $\mathcal{S}$ obtains $\mathcal{A}$'s inputs $((vk_1, sk_1), vk_2, x_1, \alpha, 1^n, cid_2)$ for the trusted party computing the functionality in phase 1. If the key vk_2 is not the same key that $\mathcal{S}$ chose for P_2, the key vk_1 in the key-pair is not the same key registered by $\mathcal{A}$, the key-pair (vk_1, sk_1) is not valid, α or n are not the same as above or x_1 is not valid for f (e.g., it is of the wrong length), then $\mathcal{S}$ sends input $\perp$ to the trusted party computing $\mathcal{F}_f^\alpha$ (as P_1's input), hands $\perp$ to $\mathcal{A}$ as its output from phase 1, outputs whatever $\mathcal{A}$ outputs and halts. Otherwise, $\mathcal{S}$ proceeds to the next step.

5. $\mathcal{S}$ chooses $cid_1 \in_R \{0,1\}^n$ and a random string r of appropriate length as in the protocol (this length is known), and sets $cid = cid_1 \| cid_2$. Then, $\mathcal{S}$ computes the cheque $\mathsf{chq}_1 = \mathsf{cheque}(cid, P_2 {\rightarrow} P_1, \alpha, r)$ (using sk_2 that it chose above) and hands chq_1 to $\mathcal{A}$ as its output from phase 1.

6. $\mathcal{S}$ receives $\mathcal{A}$'s next messages:

   (a) If $\mathcal{A}$ sends some chq_1' intended for P_2, then $\mathcal{S}$ checks that it is a valid cheque with identifier cid. If not, it ignores the message. Otherwise, it works as follows:

      i. If the value z' contained inside does not equal r as chosen by $\mathcal{S}$, then $\mathcal{S}$ outputs fail and halts.

      ii. Otherwise, $\mathcal{S}$ sends x_1 (obtained above from $\mathcal{A}$) to the trusted party computing $\mathcal{F}_f^\alpha$, receives back an output value y, and sends back fair. Then, $\mathcal{S}$ computes $\mathsf{chq}_2 = \mathsf{cheque}(cid, P_1 {\rightarrow} P_2, \alpha, y)$, using sk_1 obtained from $\mathcal{A}$ above and hands it to $\mathcal{A}$.

   (b) If $\mathcal{A}$ sends some chq_1'' intended for $\mathcal{F}_{\text{bank}}$, then $\mathcal{S}$ checks that it is a valid cheque with identifier cid. If not, it ignores the message. Otherwise, it works as follows:

      i. If the value z'' contained inside chq_1'' does not equal r as chosen by $\mathcal{S}$, then $\mathcal{S}$ outputs fail and halts.

      ii. Otherwise:

         A. If $\mathcal{S}$ already received y above, then it simulates $\mathcal{F}_{\text{bank}}$ sending P_1 the cheque chq_2 (as would occur after P_2 sends chq_2 to $\mathcal{F}_{\text{bank}}$).

         B. If not, $\mathcal{S}$ sends x_1 to the trusted party computing $\mathcal{F}_f^\alpha$, receives back the output y, and sends back fair. Then, $\mathcal{S}$ computes $\mathsf{chq}_2 = \mathsf{cheque}(cid, P_1 \rightarrow P_2, \alpha, y)$, using sk_1 obtained from $\mathcal{A}$ above and hands it to $\mathcal{A}$. Finally, $\mathcal{S}$ simulates $\mathcal{F}_{\text{bank}}$ sending P_1 the cheque chq_2 (as would occur after P_2 sends chq_2 to $\mathcal{F}_{\text{bank}}$).

7. If $\mathcal{S}$ did not send x_1 in Step 6 above, then it sends $\perp$ as P_1's input to the trusted party computing $\mathcal{F}_f^\alpha$.

8. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

This completes the simulation. First, observe that the view of $\mathcal{A}$ in such a simulation with $\mathcal{S}$ is identical to its view in a hybrid execution of Protocol π_α with an honest P_2 (and where a trusted party runs $\mathcal{F}_{CA}$ and phase 1). Furthermore, conditioned on $\mathcal{S}$ not outputting fail, the joint distribution of $\mathcal{A}$ and P_2's output in a hybrid execution is identical to that of $\mathcal{S}$ and P_2's output in an ideal execution. This is due to the fact that if $\mathcal{S}$ does not output fail, then the value output by P_2 in a hybrid execution is either $\perp$ or y, just as in an ideal execution. Furthermore, $\mathcal{S}$ works so that whenever P_2 would receive $\perp$ (resp., y) in a hybrid execution, it receives $\perp$ (resp., y) in an ideal execution (by sending $\perp$ and fair to the trusted party, appropriately). In addition, if $\mathcal{A}$ sends a valid chq_1 to $\mathcal{F}_{\text{bank}}$, then by the instructions of π_α party P_2 would send the appropriate chq_2 to $\mathcal{F}_{\text{bank}}$. This ensures that the account balances of P_1 and P_2 are unchanged, as is the case in the simulation by $\mathcal{S}$. It thus remains to show that $\mathcal{S}$ outputs fail with at most negligible probability. This is proven by a straightforward reduction to the security of the signature scheme with respect to the key vk_2. In order to see that such a reduction is possible, note that the simulation by $\mathcal{S}$ can be run in an identical way with access to a signing oracle that computes signatures with sk_2 (as provided to an adversary in the signature-forging experiment). Furthermore, if $\mathcal{S}$ outputs fail then $\mathcal{A}$ must have generated a signature that was not provided by the oracle to $\mathcal{S}$, meaning that $\mathcal{A}$ successfully forged a signature (contradicting the security of the signature scheme).

We remark that in this corruption case, the unfair command is never used (this is because P_2 receives output first and so it is never the case that $\mathcal{A}$ receives output while P_2 does not).

***P_2* is corrupted.** Let $\mathcal{A}$ be a (hybrid-model) adversary controlling party P_2. We construct a simulator $\mathcal{S}$ as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ upon its input x_2 and auxiliary-input α and n. In addition, $\mathcal{S}$ chooses keys (vk_1, sk_1) for P_1.

2. When $\mathcal{A}$ sends a $(\mathsf{Retrieve}, P_1)$ message intended for $\mathcal{F}_{CA}$, simulator $\mathcal{S}$ replies with $(\mathsf{Retrieve}, P_1, vk_1)$.

3. When $\mathcal{A}$ sends a $(\mathsf{Register}, P_2, vk_2)$ message intended for $\mathcal{F}_{CA}$, simulator $\mathcal{S}$ records vk_2.

4. $\mathcal{S}$ obtains $\mathcal{A}$'s inputs $((vk_2, sk_2), vk_1, x_2, \alpha, 1^n, cid_1, r)$ for the trusted party computing the functionality in phase 1. If the key vk_1 is not the same key that $\mathcal{S}$ chose for P_1, the key vk_2 in the key-pair is not the same as registered by $\mathcal{A}$, the key-pair (vk_2, sk_2) is not valid, α or n are not the same as above, or x_2 is not valid for f, then $\mathcal{S}$ sends the input value $\perp$ to the trusted party computing $\mathcal{F}_f^\alpha$ (as P_2's input), hands $\perp$ to $\mathcal{A}$ as its output from phase 1, outputs whatever $\mathcal{A}$ outputs and halts.

   Otherwise, $\mathcal{S}$ sends x_2 to the trusted party and receives back the output y. $\mathcal{S}$ continues with the simulation as follows:

5. $\mathcal{S}$ sets the identifier $cid = cid_1 \| cid_2$ and computes $z = r \oplus \mathsf{chq}_2$ where $\mathsf{chq}_2 = \mathsf{cheque}(cid, P_1 \to P_2, \alpha, y)$. Then, $\mathcal{S}$ computes $\mathsf{chq}_1 = \mathsf{cheque}(cid, P_2 \to P_1, \alpha, z)$ and hands chq_1 to $\mathcal{A}$ as the message it receives from P_1 in phase 2.

6. $\mathcal{S}$ receives $\mathcal{A}$'s next messages:

   (a) If $\mathcal{A}$ sends some cheque chq_2' intended for P_1, then $\mathcal{S}$ checks that it is a valid cheque with identifier cid. If not, it ignores the message. Otherwise, it works as follows:

      i. If the value z' contained inside does not equal y, then $\mathcal{S}$ outputs fail and halts.

      ii. If z' does equal y, then $\mathcal{S}$ sends fair to the trusted party computing $\mathcal{F}_f^\alpha$.

   (b) If $\mathcal{A}$ sends some cheque chq_2'' intended for $\mathcal{F}_{\text{bank}}$, then $\mathcal{S}$ checks that it is a valid cheque with identifier cid. If not, it ignores the message. Otherwise, it works as follows:

      i. If the value z'' contained inside does not equal y, then $\mathcal{S}$ outputs fail and halts.

      ii. Otherwise:

         A. If $\mathcal{S}$ already sent fair above, then it simulates $\mathcal{F}_{\text{bank}}$ sending P_2 the cheque chq_1 (as would occur after P_1 sends chq_1 to $\mathcal{F}_{\text{bank}}$).

         B. If not, $\mathcal{S}$ sends fair to the trusted party computing $\mathcal{F}_f^\alpha$ and simulates $\mathcal{F}_{\text{bank}}$ sending P_2 the cheque chq_1.

7. If $\mathcal{S}$ does not send fair in Step 6 above, then it sends unfair to the trusted party computing $\mathcal{F}_f^\alpha$.

8. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

The analysis for this case is almost identical to the previous one. Namely, there can only be a difference between an execution of π_α with $\mathcal{A}$ and the simulation with $\mathcal{S}$ in the ideal model if $\mathcal{A}$ succeeds in forging a signature. This completes the proof. ■

Sequential composition. In general, secure protocols are guaranteed to remain secure when run sequentially many times [5]. However, this is *not* necessarily the case when some joint state is kept between executions [17]. In Protocol π_α, the parties' signing and encryption keys are used in many executions and so sequential composition is not automatically guaranteed. This can be solved in two ways. One solution is to use different keys in each execution. However, since we prefer to assume a standard public-key infrastructure, the user's keys must be fixed throughout. A second solution is to ensure that the signatures in each execution are valid only in that execution by including a unique identifier inside each signature, and having the party who verifies the signature check that the identifier is correct. This solution was used in [7] and shown to achieve the necessary level of security. Observe that in Protocol π_α, the parties include random cheque identifiers cid_1 and cid_2 in order to achieve this exact effect. (Note that if these identifiers were not included, then for example P_1 can send a chq_1 from a previous execution thereby causing the output of P_2 to be that of a previous execution and not the current one.) In order to prove security under sequential composition, the only difference is that $\mathcal{S}$ may output fail even when $\mathcal{A}$ provides a signature that was provided by the signing oracle. Specifically, this can happen if a cid_i appears twice in two different executions (in such a case, $\mathcal{A}$ can provide the cheque from the previous execution, where the output may be different to y). However, since $cid_i \in_R \{0,1\}^n$ and there are only a polynomial number of executions, this can happen with at most negligible probability. Thus, Protocol π_α remains secure also under sequential composition.

Acknowledgements

We would like to thank Jonathan Katz for helpful discussions about how to formalize the notion of legally enforceable fairness, and for comments on the write-up.

References

[1] N. Asokan, M. Schunter and M. Waidner. Optimistic Protocols for Fair Exchange. In 4*th CCS*, pages 8–17, 1997.

[2] D. Beaver and S. Goldwasser. Multiparty Computation with Faulty Majority. In 30*th FOCS*, pages 468–473, 1989.

[3] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In 20*th STOC,* pages 1–10, 1988.

[4] C. Cachin and J. Camenisch. Optimistic Fair Secure Computation. In *CRYPTO 2000*, Springer-Verlag (LNCS 1880), pages 93–111, 2000.

[5] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[6] R. Canetti. Universally Composable Signature, Certification, and Authentication. In 17*th IEEE Computer Security Foundations Workshop* (CSFW), pages 219–235, 2004.

[7] R. Canetti and T. Rabin. Universal Composition with Joint State. In *CRYPTO 2003*, Springer-Verlag (LNCS 2729), pages 265–281, 2003.

[8] D. Chaum, C. Crépeau and I. Damgard. Multi-party Unconditionally Secure Protocols. In 20*th STOC*, pages 11–19, 1988.

[9] L. Chen, C. Kudla and K. Paterson. Concurrent Signatures. In *EUROCRYPT 2004*, Springer-Verlag (LNCS 3027), pages 287–305, 2004.

[10] R. Cleve. Limits on the Security of Coin Flips when Half the Processors are Faulty. In 18*th STOC,* pages 364–369, 1986.

[11] J. Garay, P. MacKenzie, M. Prabhakaran and K. Yang. Resource Fairness and Composability of Cryptographic Protocols. IN *TCC 2006*, Springer-Verlag (LNCS 3876), pages 404–428, 2006.

[12] S.D. Gordon, C. Hazay, J. Katz and Y. Lindell. Complete Fairness in Secure Two-Party Computation. Manuscript, 2007.

[13] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications.* Cambridge University Press, 2004.

[14] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In 19*th STOC,* pages 218–229, 1987.

[15] S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90,* Springer-Verlag (LNCS 537), pages 77–93, 1990.

[16] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. *Journal of Cryptology,* 18(3):247–287, 2005.

[17] Y. Lindell, A. Lysysanskaya and T. Rabin. On the Composition of Authenticated Byzantine Agreement. In the *Journal of the ACM*, 53(6):881–917, 2006.

[18] S. Micali. Secure Protocols with Invisible Trusted Parties. Presentation on Multi-Party Secure Protocols, Weizmann Institute of Science, Israel. June 1998.

[19] Silvio Micali. Simple and Fast Optimistic Protocols for Fair Electronic Exchange. In 22*nd PODC*, pages 12–19, 2003.

[20] B. Pinkas. Fair Secure Two-Party Computation. In *EUROCRYPT 2003*, Springer-Verlag (LNCS 2656), pages 87–105, 2003.

[21] A. Yao. How to Generate and Exchange Secrets. In 27*th FOCS*, pages 162–167, 1986.