Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation

Yehuda Lindell

Department of Computer Science and Applied Math,
Weizmann Institute of Science, Rehovot, Israel.
lindell@wisdom.weizmann.ac.il

Abstract. In this paper we show that any *two-party* functionality can be securely computed in a *constant number of rounds*, where security is obtained against malicious adversaries that may arbitrarily deviate from the protocol specification. This is in contrast to Yao's constant-round protocol that ensures security only in the face of semi-honest adversaries, and to its malicious adversary version that requires a polynomial number of rounds.

In order to obtain our result, we present a constant-round protocol for secure coin-tossing of polynomially many coins (in parallel). We then show how this protocol can be used in conjunction with other existing constructions in order to obtain a constant-round protocol for securely computing any two-party functionality. On the subject of coin-tossing, we also present a constant-round *perfect* coin-tossing protocol, where by "perfect" we mean that the resulting coins are guaranteed to be statistically close to uniform (and not just pseudorandom).

1 Introduction

1.1 Secure Two-Party Computation

In the setting of two-party computation, two parties, with respective private inputs x and y, wish to jointly compute a functionality $f(x,y) = (f_1(x,y), f_2(x,y))$, such that the first party receives $f_1(x,y)$ and the second party receives $f_2(x,y)$. This functionality may be probabilistic, in which case $f(x,y)$ is a random variable. Loosely speaking, the security requirements are that nothing is learned from the protocol other than the output (*privacy*), and that the output is distributed according to the prescribed functionality (*correctness*). The actual definition [14,1,5] blends these two conditions (see Section 2). This must be guaranteed even when one of the parties is adversarial. Such an adversary may be *semi-honest* in which case it correctly follows the protocol specification, yet attempts to learn additional information by analyzing the transcript of messages received during the execution. On the other hand, an adversary may be *malicious*, in which case it can arbitrarily deviate from the protocol specification.

The first general solutions for the problem of secure computation were presented by Yao [17] for the two-party case (with security against semi-honest adversaries) and Goldreich, Micali and Wigderson [13] for the multi-party case

J. Kilian (Ed.): CRYPTO 2001, LNCS 2139, pp. 171–189, 2001.

(with security even against malicious adversaries). Thus, despite the stringent security requirements placed on such protocols, wide-ranging completeness results were established, demonstrating that *any* probabilistic polynomial-time functionality can be securely computed (assuming the existence of trapdoor permutations).

Yao's protocol. In [17], Yao presented a *constant-round* protocol for securely computing any functionality, where the adversary may be semi-honest. Denote Party 1 and Party 2's respective inputs by x and y and let f be the functionality that they wish to compute (for simplicity, assume that both parties wish to receive $f(x, y)$). Loosely speaking, Yao's protocol works by having one of the parties (say Party 1) first generate an "encrypted" circuit computing $f(x, \cdot)$ and send it to Party 2. The circuit is such that it reveals nothing in its encrypted form and therefore Party 2 learns nothing from this stage. However, Party 2 can obtain the output $f(x, y)$ by "decrypting" the circuit. In order to ensure that nothing is learned beyond the output itself, this decryption must be "partial" and must reveal $f(x, y)$ only. Without going into unnecessary details, this is accomplished by Party 2 obtaining a series of keys corresponding to its input y such that given these keys and the circuit, the output value $f(x, y)$ (and only this value) may be obtained. Of course, Party 2 must obtain these keys without revealing anything about y and this can be done by running $|y|$ instances of a (semi-honest) secure 2-out-of-1 Oblivious Transfer protocol [7], which is constant-round. By running the Oblivious Transfer protocols in parallel, this protocol requires only a constant number of rounds.

Now consider what happens if Yao's protocol is run when the adversary may be malicious. Firstly, we have no guarantee that Party 1 constructed the circuit so that it correctly computes $f(x, \cdot)$. Thus, *correctness* may be violated (intuitively, this can be solved using zero-knowledge proofs). Secondly, the Oblivious Transfer protocol must satisfy the requirements for secure computation (in the face of malicious adversaries), and must maintain its security when run in parallel. We note that we know of no such (highly secure) oblivious transfer protocol that runs in a constant number of rounds. Finally, if the functionality f is *probabilistic*, then Party 1 must be forced to input a truly random string into the circuit. Thus, some type of coin-tossing protocol is also required.

Secure protocol compilation. As we have mentioned, Goldreich, Micali and Wigderson [12,13] showed that assuming the existence of trapdoor permutations, there exist protocols for securely computing any multi-party functionality, where the adversary may be malicious. They achieve this in two stages. First, they show a protocol for securely computing any functionality in the semi-honest adversarial model. Next, they construct a *protocol compiler* that takes any semi-honest protocol and "converts" it into a protocol that is secure in the malicious model. As this compiler is generic, it can be applied to *any* semi-honest protocol and in particular, to the constant-round two-party protocol of Yao. However, due to the nature of their compilation, the output protocol is no longer constant-round.

1.2 Our Results

The focus of this paper is to construct a protocol compiler such that the round-complexity of the compiled protocol is of the same order as that of the original protocol. We observe that the only component of the GMW compiler for which there does *not* exist a constant-round construction is that of coin-tossing in the well [3]. Therefore, our technical contribution is in constructing a constant-round protocol for coin-tossing in the well *of polynomially many coins*. That is, we obtain the following theorem (informally stated):

Theorem 1 (constant-round coin-tossing): *Assuming the existence of one-way functions, there exists a* constant-round *protocol for the coin-tossing functionality* (*as required by the* GMW *compiler*).

In order to construct such a constant-round protocol we introduce a technique relating to the use of commitment schemes, which we believe may be useful in other settings as well. Commitment schemes are a basic building block and are used in the construction of many protocols. Consider, for example, Blum's protocol for coin-tossing a single bit [3]. In this protocol, Party 1 sends a commitment to a random-bit; then, Party 2 replies with its own random bit and finally Party 1 decommits. The difficulty in simulating such protocols is that the simulator only knows the correct value to commit to *after* the other party sends its message. However, since the simulator is bound to its commitment, it must somehow guess the correct value *before* this message is sent. In case the messages are long (say n bits rather than a single bit or $\log n$ bits), this may be problematic. Thus, rather than decommitting, we propose to have the party reveal the committed value and then prove (in zero-knowledge) the validity of this revealed value. In a real execution, this is equivalent to decommitting, since the committing party is effectively bound to the committed value by the zero-knowledge proof. However, the simulator is able to provide a *simulated* zero-knowledge proof (rather than a real one). Furthermore, this proof remains indistinguishable from a real proof even if the revealed value is incorrect (and thus the statement is false). Therefore, the simulator can effectively "decommit" to any value it wishes and is not bound in any way by the original commitment that it sends.

Combining the constant-round protocol of Theorem 1 with other known constructions, we obtain the following theorem:

Theorem 2 *Assume the existence of one-way functions. Then, there exists a protocol compiler that given a two-party protocol Π for securely computing f in the semi-honest model produces a two-party protocol Π' that securely computes f in the malicious model, so that the number of rounds of communication in Π' is within a* constant factor *of the number of rounds of communication in Π.*

We stress that, when ignoring the "round preservation" clause, the existence of a protocol compiler is not new and has been shown in [12,13] (in fact, as we have mentioned, we use most of the components of their compiler). Our contribution is in reducing the overhead of the compiler, in terms of the round-complexity, to a constant. The main result, stated in the following theorem, is obtained by applying the compiler of Theorem 2 to the constant-round protocol of Yao.

Theorem 3 *Assuming the existence of trapdoor permutations, any two-party functionality can be securely computed in the malicious model in a* constant number of rounds.

On the subject of coin-tossing, we also present a constant-round protocol for "perfect" coin-tossing (of polynomially many coins) that guarantees that the output of the coin-tossing protocol is statistically close to uniform, and not just computationally indistinguishable.

1.3 Related Work

In the setting of multi-party computation with an honest majority, Beaver, Micali and Rogaway [2] showed that any functionality can be securely computed in a constant number of rounds, where the adversary may be malicious. Unfortunately, their technique relies heavily on the fact that a majority of the parties are honest and as such cannot be applied to the case of two-party protocols. As we have described, in this paper we establish the analogous result for the setting of *two*-party computation.

1.4 Organization

In Section 2 we present the definition of secure two-party computation. Then, in Section 3 we discuss the protocol compiler of GMW and observe that in order to achieve "round-preserving" compilation, one needs only to construct a constant-round coin-tossing protocol. Our technical contribution in this paper thus begins in Section 4 where we present such a constant-round coin-tossing protocol. Finally, in Section 5 we show how perfect coin-tossing can be achieved.

2 Definitions – Secure Computation

In this section we present the definition of secure two-party computation. Our presentation is based on [9], which in turn follows [14,1,5]. We first introduce the following notation: U_n denotes the uniform distribution over $\{0,1\}^n$; for a set S we denote $s \in_R S$ when s is chosen uniformly from S; finally, computational indistinguishability is denoted by $\stackrel{\text{c}}{\equiv}$ and statistical closeness by $\stackrel{\text{s}}{\equiv}$.

Two-party computation. A two-party protocol problem is cast by specifying a random process that maps pairs of inputs to pairs of outputs (one for each party). We refer to such a process as a functionality and denote it $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$, where $f = (f_1, f_2)$. That is, for every pair of inputs (x, y), the output-pair is a random variable $(f_1(x,y), f_2(x,y))$ ranging over pairs of strings. The first party (with input x) wishes to obtain $f_1(x,y)$ and the second party (with input y) wishes to obtain $f_2(x,y)$. We often denote such a functionality by $(x,y) \mapsto (f_1(x,y), f_2(x,y))$. Thus, for example, the basic coin-tossing functionality is denoted by $(1^n, 1^n) \mapsto (U_n, U_n)$.

Adversarial behavior. Loosely speaking, the aim of a secure two-party protocol is to protect an honest party against dishonest behavior by the other party. This "dishonest behavior" can manifest itself in a number of ways; in particular, we focus on what are known as *semi-honest* and *malicious* adversaries. A semi-honest adversary follows the prescribed protocol, yet attempts to learn more information than "allowed" from the execution. Specifically, the adversary may record the entire message transcript of the execution and attempt to learn something beyond the protocol output. On the other hand, a malicious adversary may arbitrarily deviate from the specified protocol. When considering malicious adversaries, there are certain undesirable actions that cannot be prevented. Specifically, a party may refuse to participate in the protocol, may substitute its local input (and enter with a different input) and may abort the protocol prematurely.

Security of protocols (informal). The security of a protocol is analyzed by comparing what an adversary can do in the protocol to what it can do in an ideal scenario that is secure by definition. This is formalized by considering an *ideal* computation involving an incorruptible *trusted third party* to whom the parties send their inputs. The trusted party computes the functionality on the inputs and returns to each party its respective output. Loosely speaking, a protocol is secure if any adversary interacting in the real protocol (where no trusted third party exists) can do no more harm than if it was involved in the above-described ideal computation.

Execution in the ideal model. The ideal model differs for semi-honest and malicious parties. First, for semi-honest parties, an ideal execution involves each party sending their respective input to the trusted party and receiving back their prescribed output. An honest party then outputs this output, whereas a semi-honest party may output an arbitrary (probabilistic polynomial-time computable) function of its initial input and the message it obtained from the trusted party. (See [9] for a formal definition of the ideal and real models for the case of semi-honest adversaries.)

We now turn to the ideal model for malicious parties. Since some malicious behavior cannot be prevented (for example, early aborting), the definition of the ideal model in this case is somewhat more involved. An ideal execution proceeds as follows:

Inputs: Each party obtains an input, denoted z.

Send inputs to trusted party: An honest party always sends z to the trusted party. A malicious party may, depending on z, either abort or sends some $z' \in \{0,1\}^{|z|}$ to the trusted party.

Trusted party answers first party: In case it has obtained an input pair, (x, y), the trusted party (for computing f), first replies to the first party with $f_1(x, y)$. Otherwise (i.e., in case it receives only one input), the trusted party replies to both parties with a special symbol, $\perp$.

Trusted party answers second party: In case the first party is malicious it may, depending on its input and the trusted party's answer, decide to *stop* the trusted party. In this case the trusted party sends $\perp$ to the second party.

Otherwise (i.e., if not stopped), the trusted party sends $f_2(x, y)$ to the second party.

Outputs: An honest party always outputs the message it has obtained from the trusted party. A malicious party may output an arbitrary (probabilistic polynomial-time computable) function of its initial input and the message obtained from the trusted party.

Let $f : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^* \times \{0,1\}^*$ be a functionality, where $f = (f_1, f_2)$, and let $\overline{M} = (M_1, M_2)$ be a pair of families of non-uniform probabilistic *expected polynomial-time* machines (representing parties in the ideal model). Such a pair is admissible if for at least one $i \in \{1, 2\}$ we have that M_i is honest. Then, the joint execution of f under $\overline{M}$ in the ideal model (on input pair (x, y)), denoted $\mathsf{ideal}_{f,\overline{M}}(x, y)$, is defined as the output pair of M_1 and M_2 from the above ideal execution. For example, in the case that M_1 is malicious and always aborts at the outset, the joint execution is defined as $(M_1(x, \perp), \perp)$. Whereas, in case M_1 never aborts, the joint execution is defined as $(M_1(x, f_1(x', y)), f_2(x', y))$ where $x' = M_1(x)$ is the input that M_1 gives to the trusted party.

Execution in the real model. We next consider the real model in which a real (two-party) protocol is executed (and there exists no trusted third party). In this case, a malicious party may follow an arbitrary feasible strategy; that is, any strategy implementable by non-uniform expected polynomial-time machines. In particular, the malicious party may abort the execution at any point in time (and when this happens prematurely, the other party is left with no output).

Let f be as above and let Π be a two-party protocol for computing f. Furthermore, let $\overline{M} = (M_1, M_2)$ be a pair of families of non-uniform probabilistic *expected polynomial-time* machines (representing parties in the real model). Such a pair is admissible if for at least one $i \in \{1, 2\}$ we have that M_i is honest (i.e., follows the strategy specified by Π). Then, the joint execution of Π under $\overline{M}$ in the real model (on input pair (x, y)), denoted $\mathsf{real}_{\Pi,\overline{M}}(x, y)$, is defined as the output pair of M_1 and M_2 resulting from the protocol interaction.

Security as emulation of a real execution in the ideal model. Having defined the ideal and real models, we can now define security of protocols. Loosely speaking, the definition asserts that a secure two-party protocol (in the real model) emulates the ideal model (in which a trusted party exists). This is formulated by saying that admissible pairs in the ideal model are able to simulate admissible pairs in an execution of a secure real-model protocol.

Definition 4 (security in the malicious model): *Let f and Π be as above. Protocol Π is said to* securely compute *f* (in the malicious model) *if there exists a probabilistic polynomial-time computable transformation of pairs of admissible families of non-uniform probabilistic expected polynomial-time machines $\overline{A} = (A_1, A_2)$ for the real model into pairs of admissible families of non-uniform probabilistic expected polynomial-time machines $\overline{B} = (B_1, B_2)$ for the ideal model such that*

$$\{\mathsf{ideal}_{f,\overline{B}}(x, y)\}_{x,y \text{ s.t. } |x|=|y|} \stackrel{\text{c}}{\equiv} \{\mathsf{real}_{\Pi,\overline{A}}(x, y)\}_{x,y \text{ s.t. } |x|=|y|}$$

Remark: The above definition is different from the standard definition in that the adversary (in both the ideal and real models) is allowed to run in *expected* polynomial-time (rather than *strict* polynomial-time). This seems to be inevitable given that currently known constant-round zero-knowledge proofs require *expected* polynomial-time simulation. We stress that an honest party always runs in *strict* polynomial time.

3 Two-Party Computation Secure Against Malicious Adversaries

3.1 The Compiler of Goldreich, Micali, and Wigderson [13]

Goldreich, Micali and Wigderson [13] showed that assuming the existence of trapdoor permutations, there are secure protocols (in the malicious model) for any multi-party functionality. Their methodology works by first presenting a protocol secure against semi-honest adversaries. Next, a *compiler* is applied that transforms *any* protocol secure against semi-honest adversaries into a protocol secure against malicious adversaries. Thus, their compiler can also be applied to the constant-round two-party protocol of Yao [17] (as it is secure against semi-honest adversaries). However, as we shall see, the output protocol itself is *not* constant-round. In this section, we describe the [13] compiler and show what should be modified in order to obtain a *constant-round* compiler instead.

Enforcing semi-honest behavior. The GMW compiler takes for input a protocol secure against semi-honest adversaries; from here on we refer to this as the "basic protocol". Recall that this protocol is secure in the case that each party follows the protocol specification exactly, using its input and uniformly chosen random tape. Thus, in order to obtain a protocol secure against malicious adversaries, we need to enforce potentially malicious parties to behave in a semi-honest manner. First and foremost, this involves forcing the parties to follow the prescribed protocol. However, this only makes sense relative to a *given* input and random tape. Furthermore, a malicious party must be forced into using a *uniformly chosen* random tape. This is because the security of the basic protocol may depend on the fact that the party has no freedom in setting its own randomness.

An informal description of the GMW compiler. In light of the above discussion, the compiler begins by having each party commit to its input. Next, the parties run a coin-tossing protocol in order to fix their random tapes (clearly, this protocol must be secure against malicious adversaries). A regular coin-tossing protocol in which both parties receive the same uniformly distributed string does not help us here. This is because the parties' random tapes must remain secret. This is solved by augmenting the coin-tossing protocol so that one party receives a uniformly distributed string (to be used as its random tape) and the other party receives a commitment to that string. Now, following these two steps, each party holds its own uniformly distributed random-tape and a commitment to the other party's input and random-tape. Therefore, each party can be "forced" into working consistently with this specific input and random-tape.

We now describe how this behavior is enforced. A protocol specification is a deterministic function of a party's view consisting of its input, random tape and messages received so far. As we have seen, each party holds a commitment to the input and random tape of the other party. Furthermore, the messages sent so far are public. Therefore, the assertion that a new message is computed according to the protocol is of the $\mathcal{NP}$ type (and the party sending the message knows an adequate $\mathcal{NP}$-witness to it). Thus, the parties can use zero-knowledge proofs to show that their steps are indeed according to the protocol specification. As the proofs used are zero-knowledge, they reveal nothing. On the other hand, due to the soundness of the proofs, even a malicious adversary cannot deviate from the protocol specification without being detected. We thus obtain a reduction of the security in the malicious case to the given security of the basic protocol against semi-honest adversaries.

In summary, the components of the compiler are as follows (from here on "secure" refers to security against malicious adversaries):

1. **Input Commitment:** In this phase the parties execute a secure protocol for the following functionality:

   $$((x, r), 1^n) \mapsto (\lambda, C(x; r))$$

   where x is the party's input string (and r is the randomness chosen by the committing party).
   A secure protocol for this functionality involves the committing party sending $C(x; r)$ to the other party followed by a zero-knowledge proof of knowledge of (x, r). Note that this functionality ensures that the committing party knows the value being committed to.
2. **Coin Generation:** The parties generate t-bit long random tapes (and corresponding commitments) by executing a secure protocol in which one party receives a commitment to a uniform string of length t and the other party receives the string itself (to be used as its random tape) and the decommitment (to be used later for proving "proper behavior"). That is, the parties compute the functionality:

   $$(1^n, 1^n) \mapsto ((U_t, U_{t \cdot n}), C(U_t; U_{t \cdot n}))$$

   (where we assume that to commit to a t-bit string, C requires $t \cdot n$ random bits).
3. **Protocol Emulation:** In this phase, the parties run the basic protocol whilst proving (in zero-knowledge) that their steps are consistent with their input string, random tape and prior messages received.

A detailed description of each phase of the compiler and a full proof that the resulting protocol is indeed secure against malicious adversaries can be found in [9].

3.2 Achieving Round-Preserving Compilation

As we have mentioned, our aim in this work is to show that the GMW compiler can be implemented so that the number of rounds in the resulting compiled

protocol is within a *constant factor* of the number of rounds in the original semi-honest protocol. We begin by noting that using currently known constructions, Phases 1 and 3 of the GMW compiler can be implemented in a constant number of rounds. That is,

Proposition 5 *Assuming the existence of one-way functions, both the* input-commitment *and* protocol-emulation *phases can be securely implemented in a* constant number of rounds.

First consider the input-commitment phase. As mentioned above, this phase can be securely implemented by having the committing party send a perfectly binding commitment of its input to the other party, followed by a zero-knowledge proof of knowledge of the committed value. Both constant-round commitment schemes and constant-round zero-knowledge arguments of knowledge are known to exist by the works of Naor [15] and Feige and Shamir [8], respectively (these constructions can also be based on any one-way function). Thus the input-commitment phase can be implemented as required for Proposition 5.[1] Next, we recall that a secure implementation of the protocol emulation phase requires zero-knowledge proofs for $\mathcal{NP}$ only. Thus, once again, using the argument system of [8], this can be implemented in a constant number of rounds (using any one-way function).

Constant-round coin tossing. In contrast to the input-commitment and protocol-emulation phases of the GMW compiler, known protocols for tossing polynomially many coins do not run in a constant number of rounds. Rather, single coins are tossed sequentially (and thus $poly(n)$ rounds are needed). In particular, the proof of [9] does *not* extend to the case that many coins are tossed in parallel. Thus, in order to obtain a round-preserving compiler, it remains to present a secure protocol for the coin-generation functionality that works in a constant number of rounds. Furthermore, it is preferable to base this protocol on the existence of one-way functions only (so that this seemingly minimal assumption is all that is needed for the entire compiler). In the next section we present such a coin-tossing protocol, thereby obtaining Theorem 2 (as stated in the introduction).

3.3 Constant-Round Secure Computation

Recall that by Yao [17], assuming the existence of trapdoor permutations, any two-party functionality can be securely computed in the *semi-honest* model in a constant-number of rounds. Thus, applying the constant-round compiler of Theorem 2 to Yao's protocol, we obtain a constant-round protocol that is secure

[1] We note that the protocol for the commit-functionality, as described in [9], is for a single-bit only (and thus the compiler there runs this protocol sequentially for each bit of the input). However, the proof for the commit-functionality remains almost identical when the functionality is extended to commitments of $poly(n)$-bit strings (rather than for just a single-bit).

in the *malicious* model, and prove Theorem 3. That is, assuming the existence of trapdoor permutations, any two-party functionality can be securely computed in the *malicious* model in a *constant-number of rounds*.

4 The Augmented Coin-Tossing Protocol

4.1 The Augmented Coin-Tossing Functionality

In this section we present our coin-tossing protocol, thus proving Theorem 1. In a basic coin-tossing functionality, both parties receive identical uniformly distributed strings. That is, the functionality is defined as: $(1^n, 1^n) \mapsto (U_m, U_m)$ for some $m = poly(n)$. This basic coin-tossing is augmented as follows. Let F be any deterministic function. Then, define the augmented coin-tossing functionality by:

$$(1^n, 1^n) \mapsto (U_m, F(U_m))$$

That is, the first party indeed receives a uniformly distributed string. However, the second party receives F applied to that string (rather than the string itself). Setting F to the *identity function*, we obtain basic coin-tossing. However, recall that the coin-generation component of the GMW compiler requires the following functionality:

$$(1^n, 1^n) \mapsto ((U_t, U_{t \cdot n}), C(U_t; U_{t \cdot n}))$$

where C is a commitment scheme (and we assume that C requires n random bits for every bit committed to). Then, this functionality can be realized with our augmentation by setting $m = t + t \cdot n$ and $F(U_m) = C(U_t; U_{t \cdot n})$. Thus, the second party receives a commitment to a uniformly distributed string of length t and the first party receives the string and its decommitment. Recall that in the compiler, the party uses the t-bit string as its random tape and the decommitment in order to prove in zero-knowledge that it is acting consistently with this random tape (and its input).

4.2 Motivating Discussion

In order to motivate our construction of a constant-round coin-tossing protocol, we consider the special case of basic coin-tossing (i.e., where F is the identity function). A natural attempt at a coin-tossing protocol follows:

Protocol 1 (Attempt at Basic Coin-Tossing):

1. *Party 1 chooses a random string* $s_1 \in_R \{0,1\}^m$ *and sends* $c = \text{Commit}(s_1) = C(s_1; r)$ *(where* r *is randomly chosen).*
2. *Party 2 chooses a random string* $s_2 \in_R \{0,1\}^m$ *and sends it to Party 1.*
3. *Party 1 decommits to* s_1 *sending the pair* (s_1, r).

Party 1 always outputs $s \stackrel{\text{def}}{=} s_1 \oplus s_2$*, whereas Party 2 outputs* $s_1 \oplus s_2$ *if Party 1's decommitment is correct and* $\perp$ *otherwise.*

We note that when $m = 1$ (i.e., a single bit), the above protocol is the basic coin-tossing protocol of Blum [3] (a formal proof of the security of this protocol can be found in [9]). However, here we are interested in a parallelized version where the parties attempt to simultaneously generate an m-bit random string (for any $m = poly(n)$). Intuitively, due to the secrecy of the commitment scheme, the string s_2 chosen by (a possibly malicious) Party 2 cannot be dependent on the value of s_1. Thus if s_1 is chosen uniformly, the resulting string $s = s_1 \oplus s_2$ is close to uniform. On the other hand, consider the case that Party 1 may be malicious. Then, by the protocol, Party 1 is committed to s_1 before Party 2 sends s_2. Thus, if s_2 is chosen uniformly, the string $s = s_1 \oplus s_2$ is uniformly distributed. We note that due to the binding property of the commitment scheme, Party 1 cannot alter the initial string committed to. We conclude that neither party is able to bias the output string.

However, the infeasibility of either side to bias the resulting string is not enough to show that the protocol is secure. This is because the definition of secure computation requires that the protocol simulate an *ideal execution* in which a trusted third party chooses a random string s and gives it to both parties. Loosely speaking, this means that there exists a simulator that works in the ideal model and simulates an execution with a (possibly malicious) party such that the joint output distribution (in this ideal scenario) is indistinguishable from when the parties execute the real protocol.

Protocol 1 seems not to fulfill this more stringent requirement. That is, our problem in proving the security of Protocol 1 is with constructing the required simulator. The main problem that occurs is regarding the simulation of Party 2.

Simulating a malicious Party 2: The simulator receives a uniformly distributed string s and must generate an execution consistent with s. That is, the commitment $c = C(s_1)$ given by the simulator to Party 2 must be such that $s_1 \oplus s_2 = s$ (where s_2 is the string sent by Party 2 in Step 2 of the protocol). However, s_1 is chosen and fixed (via a perfectly binding commitment) *before* s_2 is chosen by Party 2. Since the commitment is perfectly binding, even an all-powerful simulator cannot "cheat" and decommit to a different value. This problem is compounded by the fact that Party 2 may choose s_2 based on the commitment received to s_1 (by say invoking a pseudorandom function on c). Therefore, rewinding Party 2 and setting s_1 to equal $s \oplus s_2$ will not help (as s_2 will change and thus once again $s_1 \oplus s_2$ will equal s with only negligible probability). We note that this problem does not arise in the single-bit case as there are only two possible values for s_2 and thus the simulator succeeds with probability $1/2$ each time.

A problem relating to abort: The above problem arises even when the parties *never* abort. However, another problem in simulation arises due to the ability of the parties to abort. In particular, simulation of Party 1 in Protocol 1 is easy assuming that Party 1 never aborts. On the other hand, when Party 1's abort probability is unknown (and specifically when it is neither negligible nor noticeable), we do not know how to construct a simulator that does not skew the real probability of abort in the simulated execution. Once again, this problem

is considerably easier in the single-bit case since Party 1's decision of whether or not to abort is based on only a single bit sent by Party 2 in Step 2 of the protocol (and so there are only three possible probabilities).

We note that basic coin-tossing is a special case of the augmented coin-tossing functionality. Thus, the same problems (and possibly others) must be solved in order to obtain an augmented coin-tossing protocol. As we will show, our solutions for these problems are enough for the augmented case as well.

Evidence that Protocol 1 is not secure: One may claim that the above *3-round* protocol may actually be secure and that the above-described difficulties are due to our proof technique. However, it can be shown that if there exists a secure 3-round protocol for coin-tossing (where the simulation uses black-box access to the malicious party), then there exist 3-round black-box zero-knowledge arguments for $\mathcal{NP}$. By [11], this would imply that $\mathcal{NP} \subseteq \mathcal{BPP}$. We note that all known simulations of secure protocols are indeed black-box.

4.3 The Actual Protocol

Before presenting the protocol itself, we discuss how we solve the problems described in the above motivating discussion.

- *Party 1 is malicious:* As described, when Party 1 is malicious, the problem that arises is that of aborting. In particular, Party 1 may decide to abort depending on the string s_2 sent to it by Party 2. This causes a problem in ensuring that the probability of abort in the simulation is negligibly close to that in a real execution. This is solved by having Party 1 send a proof of knowledge of s_1 after sending the commitment. Then, the simulator can extract s_1 from the proof of knowledge and can send $s_2 = s_1 \oplus s$ (where s is the string chosen by the trusted party) without waiting for Party 1 to decommit in a later step.

- *Party 2 is malicious:* As described, the central problem here is that Party 1 must commit itself to s_1 before s_2 is known (yet $s_1 \oplus s_2$ must equal s). This cannot be solved by rewinding because Party 2 may choose s_2 based on the commitment to s_1 that it receives (and thus changing the commitment changes the value of s_2). We solve this problem by not having Party 1 decommit at all; rather, it sends $s = s_1 \oplus s_2$ (or $F(s_1 \oplus s_2)$ in the augmented case) and proves in zero-knowledge that the value sent is consistent with its commitment and s_2. Thus, the simulator (who can generate proofs to false statements of this type) is able to "cheat" and send s (or $F(s)$) irrespective of the *real* value committed to in Step 1.[2]

  This technique of not decommitting, but rather revealing the committed value and proving (in zero-knowledge) that this value is correct, is central to

[2] In general, nothing can be said about a simulated proof of a false statement. However, in the specific case of statements regarding commitment values, proofs of false statements are indistinguishable from proofs of valid statements. This is due to the hiding property of the commitment scheme.

our simulation strategy. Specifically, it enables us to "decommit" to a value that is unknown at the time of the commitment. (As we have mentioned, in order for the simulation to succeed, Party 2 must be convinced that the commitment of Step 1 is to s_1, where $s_1 \oplus s_2 = s$. However, the correct value of s_1 is only known to the simulator after Step 2.)

We now present our constant-round protocol for the augmented secure coin-tossing functionality: $(1^n, 1^n) \mapsto (U_m, F(U_m))$, for $m = poly(n)$. For the sake of simplicity, our presentation uses a non-interactive commitment scheme (which is easily constructed given any 1–1 one-way function). However, the protocol can easily be modified so that an interactive commitment scheme is used instead (in particular, the two-round scheme of Naor [15]).

Protocol 2 (Augmented Parallel Coin-Tossing):

1. *Party 1 chooses* $s_1 \in_R \{0,1\}^m$ *and sends* $c = C(s_1; r)$ *for a random* r *to Party 2 (using a perfectly binding commitment scheme).*
2. *Party 1 proves knowledge of* (s_1, r) *with a (constant round) zero-knowledge argument of knowledge with negligible error. If the proof fails, then Party 2 aborts with output* $\perp$.
3. *Party 2 chooses* $s_2 \in_R \{0,1\}^m$ *and sends* s_2 *to Party 1.*
4. *If until this point Party 1 received an invalid message from Party 2, then Party 1 aborts, outputting* $\perp$.
   Otherwise, Party 1 sends $y = F(s_1 \oplus s_2)$.
5. *Party 1 proves to Party 2 using a (constant round) zero-knowledge argument that there exists a pair* (s_1, r) *such that* $c = C(s_1; r)$ *and* $y = F(s_1 \oplus s_2)$ *(that is, Party 1 proves that* y *is consistent with* c *and* s_2*).*[3] *If the proof fails, then Party 2 aborts with output* $\perp$.
6. Output:
   - *Party 1 outputs* $s_1 \oplus s_2$ *(even if Party 2 fails to correctly complete the verification of the proof in Step 5).*
   - *Party 2 outputs* y.

Round complexity: Using the constant-round zero-knowledge argument system of Feige and Shamir [8] and the constant-round commitment scheme of Naor [15], Protocol 2 requires a constant number of rounds only. We note that the proof system of [8] is also a proof of knowledge.

[3] It may appear that the reason that Party 1 does not decommit to c is due to the fact that Party 2 should only learn $F(s)$, and not s itself (if Party 1 decommits, then s is clearly revealed). Following this line of thinking, if F was the identity function, then Steps 4 and 5 could be replaced by Party 1 sending the actual decommitment. However, we stress that we do not know how to prove the security of such a modified protocol. The fact that Party 1 does not decommit, even when F is the identity function, is crucial to our proof of security.

Sufficient assumptions: All the components of Protocol 2 can be implemented using any one-way function. In particular the string commitment of Naor [15] can be used (this requires an additional pre-step in which Party 2 sends a random string to Party 1; however this step is of no consequence to the proof). Furthermore, the zero-knowledge argument of knowledge of [8] can be used in both Steps 2 and 5. Since both the [15] and [8] protocols only assume the existence of one-way functions, this is the only assumption required for the protocol.

Theorem 6 *Assuming the existence of one-way functions, Protocol 2 is a secure protocol for augmented parallel coin-tossing.*

Proof: We need to show how to efficiently transform any admissible pair of machines (A_1, A_2) for the real model into an admissible pair of machines (B_1, B_2) for the ideal model. We denote the trusted third party by T, the coin-tossing functionality by f and Protocol 2 by Π. We first consider the case that A_1 is adversarial.

Lemma 7 *Let (A_1, A_2) be an admissible pair of probabilistic expected polynomial-time machines for the real model in which A_2 is* honest. *Then, there exists an efficient transformation of (A_1, A_2) into an admissible pair of probabilistic expected polynomial-time machines (B_1, B_2) for the ideal model such that*

$$\{\mathsf{ideal}_{f,\overline{B}}(1^n, 1^n)\}_{n\in\mathsf{N}} \stackrel{c}{\equiv} \{\mathsf{real}_{\Pi,\overline{A}}(1^n, 1^n)\}_{n\in\mathsf{N}}$$

Proof Sketch: In this case the second party is honest and thus B_2 is determined. We now briefly describe the transformation of the real-model adversary A_1 into an ideal-model adversary B_1. Machine B_1 emulates an execution of A_1 with A_2 by playing the role of (an honest party) A_2 in most of the execution. (In particular, B_1 verifies the zero-knowledge proofs provided by A_1 and "checks" that A_1 is not cheating.) However, instead of randomly choosing the string s_2 in Step 3 (as A_2 would), machine B_1 first obtains the value s_1 (committed to by A_1) by running the extractor for the proof of knowledge of Step 2. Then, B_1 sets $s_2 = s_1 \oplus s$ where s is the output provided by the trusted third party.

It is easy to see that if A_1 follows the instructions of Protocol 2, then the output distributions in the ideal and real models are identical. This is because A_1's view in the ideal-model emulation with B_1 is identical to that of a real execution with A_2. Furthermore, since $s_1 \oplus s_2 = s$, the result of the execution is consistent with the outputs chosen by the trusted third party. However, A_1 may not follow the instructions of the protocol and nevertheless we must show that the real and ideal output distributions remain computationally indistinguishable (in fact, they are even *statistically close*). This can be seen by noticing that differences between the ideal and real executions can occur only if the extraction fails even though A_1 succeeded in proving the proof of Step 2, or if A_1 successfully cheats in the zero-knowledge proof of Step 5. Since both of these events occur with at most negligible probability, we have that the distributions are statistically close. ■

We now consider the case that A_2 is adversarial.

Lemma 8 *Let (A_1, A_2) be an admissible pair of probabilistic expected polynomial-time machines for the real model in which A_1 is* honest. *Then, there exists an efficient transformation of (A_1, A_2) into an admissible pair of probabilistic expected polynomial-time machines (B_1, B_2) for the ideal model such that*

$$\{\mathsf{ideal}_{f,\overline{B}}(1^n, 1^n)\}_{n\in\mathsf{N}} \stackrel{\mathrm{c}}{\equiv} \{\mathsf{real}_{\Pi,\overline{A}}(1^n, 1^n)\}_{n\in\mathsf{N}}$$

Proof Sketch: In this case the first party is honest and thus B_1 is determined. We now transform the real-model adversary A_2 into an ideal-model adversary B_2, where the transformation is such that B_2 uses black-box access to A_2. Specifically, B_2 chooses a uniform random tape, denoted R, for A_2 and invokes A_2 on input 1^n and random tape R. Once the input and random tape are fixed, A_2 is a deterministic function of messages received during a protocol execution. Thus $A_2(1^n, R, \overline{m})$ denotes the message sent by A_2 with input 1^n, random-tape R and sequence $\overline{m}$ of incoming messages to A_2.

The transformation works by having B_2 emulate an execution of A_2, while playing A_1's role. Machine B_2 does this when interacting with the trusted third party T and its aim is to obtain an execution with A_2 that is consistent with the output received from T. Therefore, B_2 has both external communication with T and "internal", emulated communication with A_2. Machine B_2 works as follows:

1. The ideal adversary B_2 chooses a uniformly distributed random tape R for the real adversary A_2, invokes $A_2(1^n, R)$ and (internally) passes to A_2 the commitment $c = C(0^m; r)$ for a random r (recall that in a real execution, A_2 expects to receive $C(s_1; r)$ for a random s_1).
2. B_2 invokes the *simulator* for the zero-knowledge argument of knowledge of the decommitment of c, using $A_2(1^n, R, c)$ as the verifier. (That is, this is a simulation of the proof of knowledge that A_1 is supposed to prove to A_2 in a real execution.)
3. B_2 obtains s_2 from A_2. (Recall that this is formally stated by having B_2 compute the function $A_2(1^n, R, c, t_{pok})$, where t_{pok} is the resulting transcript from the zero-knowledge proof of knowledge simulation).
   If at any point until here A_2 sent an invalid message, then B_2 aborts and outputs $A_2(1^n, R, c, t_{pok})$.
4. The ideal adversary B_2 sends 1^n to the (external) trusted third party T and receives the output $F(s)$.
   Next, B_2 (internally) passes to A_2 the string $y = F(s)$.
5. B_2 invokes the simulator for the zero-knowledge proof of Step 5 of the Protocol with the verifier role being played by $A_2(1^n, R, c, t_{pok}, y)$. Denote the transcript from the simulation of the zero-knowledge proof by t_{pf}.
6. B_2 outputs $A_2(1^n, R, c, t_{pok}, y, t_{pf})$.

We need to show that

$$\{\mathsf{ideal}_{f,\overline{B}}(1^n, 1^n)\}_{n\in\mathsf{N}} \stackrel{\mathrm{c}}{\equiv} \{\mathsf{real}_{\Pi,\overline{A}}(1^n, 1^n)\}_{n\in\mathsf{N}}$$

The following differences are evident between the ideal and real executions:

- The commitment received by A_2 (in the internal emulation by B_2) is to 0^m, rather than to a random string consistent with $y = F(s)$ and s_2 (as is the case in a real execution). However, by the indistinguishability of commitments, this should not make a difference.
- In the internal emulation by B_2, the zero-knowledge proofs are simulated and not real proofs. However, by the indistinguishability of simulated proofs, this should also not make a difference. As mentioned above, this holds even though the statement "proved" by B_2 in Step 5 is false with overwhelming probability.

The natural way to proceed at this point would be to define a hybrid experiment in which the commitment given by B_2 to A_2 is to s_1 and yet the zero-knowledge proofs are simulated. (In this hybrid experiment, s_1 must be such that $y = F(s_1 \oplus s_2)$.) However, such a hybrid experiment is problematic because the value of s_1 that is consistent with both y (from T) and s_2 is *unknown* at the point that B_2 generates the commitment. We must therefore bypass this problem before defining the hybrid experiment. We do this by defining the following *mental experiment* with a modified party B_2' (replacing Step 4 only of B_2 above):

4′. B_2' chooses $s_1 \in_R \{0,1\}^m$ (independently of what it has previously seen) and computes $y = F(s_1 \oplus s_2)$ (rather than obtaining $y = F(s)$ from T). Next, B_2' (internally) passes A_2 the string y.

Notice that B_2' *does not* interact with any trusted third party at all. Rather, it chooses a uniformly distributed s, and computes $F(s)$ itself (observe that choosing s_1 uniformly and setting $s = s_1 \oplus s_2$ is equivalent to uniformly choosing s). We stress that B_2' does not work in the ideal model, but is rather a mental experiment. Despite this, since B_2' chooses s_1 independently of what it has seen, the distribution generated by B_2' is identical to that of the ideal model (where s is chosen by the trusted party).

Next, notice that if we move the step in which s_1 is chosen to before the first step of B_2', then this makes no difference to the output distribution. Having done this, it is possible for B_2' to send a commitment to s_1 rather than to 0^m. Thus, the above-described hybrid experiment can be defined. That is, we define a hybrid setting (with a party B_2'') in which B_2'' initially sends a commitment to s_1 (rather than to 0^m). Thus, in terms of the commitment, the hybrid experiment is identical to a real execution (and different to the mental experiment and ideal model). On the other hand, the zero-knowledge proofs in the hybrid experiment are simulated (as in the mental experiment), rather than being actual proofs (as in the real model). Then, the indistinguishability of the mental experiment from the real model is demonstrated by first showing the indistinguishability of the the hybrid and mental experiments (where the only difference is regarding the initial commitment) and then showing the indistinguishability of the hybrid and real executions (where the only difference is regarding the simulated zero-knowledge proofs). Since the output of an ideal-model execution is identically distributed to the output from the mental experiment, this completes the proof. ■

This completes the proof of Theorem 6. ■

4.4 Comparing Protocol 2 to the Protocol of [9]

The protocol for augmented coin-tossing presented by Goldreich [9] is for tossing a single bit only (i.e., where $m = 1$). Thus, in order to toss polynomially many coins, Goldreich suggests running the single-bit protocol many times sequentially. However, the only difference between Protocol 2 and the protocol of [9] is that here m can be any value polynomial in n and there m is fixed at 1 (i.e., by setting $m = 1$ in our protocol, we obtain the exact protocol of [9]). Despite this, our *proof* is different and works for any $m = poly(n)$ whereas the *proof* of [9] relies heavily on $m = 1$ (or at the most $m = O(\log n)$).[4] Furthermore, there is a conceptual difference in the role of the two zero-knowledge proofs in the protocol. In [9], these proofs are used in order to obtain *augmented* coin-tossing (and are not needed for the case that F is the identity function). However, here these proofs are used for obtaining coin-tossing of $m = poly(n)$ coins in parallel, even when F is the identity function.

5 Perfect Coin-Tossing

In this section we present a constant-round protocol for *perfect* coin tossing. By perfect coin tossing, we mean that the output distribution of a real execution is *statistically close* to the output distribution of the ideal process (rather than the distributions being only computationally indistinguishable as required by secure computation); see Theorem 9. As in the previous section, the functionality we consider is that of augmented coin tossing: $(1^n, 1^n) \mapsto (U_m, F(U_m))$.

The protocol is almost identical to Protocol 2 except that the commitment scheme used is perfectly hiding and the zero-knowledge arguments are perfect. These primitives are known to exist assuming the existence of families of clawfree functions or collision-resistant hash functions. Thus we rely here on a (seemingly) stronger assumption than merely the existence of one-way functions. We note that Protocol 1 is a protocol for *perfect coin tossing* of a single bit and thus perfect coin tossing of m coins can be achieved in $O(m)$ rounds (see the proof in [9] which actually demonstrates statistical closeness). In this section we show that perfect coin tossing of polynomially many coins can also be achieved in a *constant number of rounds*.

Protocol 3 (Augmented Perfect Coin-Tossing):
An augmented perfect coin-tossing protocol is constructed by taking Protocol 2 and making the following modifications:

- *The commitment sent by Party 1 in Step 1 is* perfectly hiding.
- *The proof of knowledge provided by Party 1 in Step 2 is* perfect *zero-knowledge.*
- *The proof provided by Party 1 in Step 5 is a* perfect *zero-knowledge* proof of knowledge. *(Recall that in Protocol 2, this proof need not be a proof of knowledge.)*

[4] In private communication, Goldreich stated that he did not know whether or not his protocol [9] can be parallelized.

Constant-round perfect zero-knowledge arguments of knowledge are known to exist assuming the existence of constant-round perfectly hiding commitment schemes [4,8]. Furthermore, constant-round perfectly-hiding commitment schemes can be constructed using families of clawfree [10] or collision-resistant hash functions [16,6]. These commitment schemes work by having the receiver first uniformly choose a function f from the family designated in the protocol. The receiver then sends f to the sender who uses it to commit to a string by sending a single message. Thus, using such a scheme, Protocol 3 begins by Party 2 choosing a function f from a clawfree or collision-resistant family and sending it to Party 1. Then, Party 1 commits using f.

We stress the use of arguments of knowledge for *both* proofs here, whereas in Protocol 2 the proof of Step 5 need not be a proof of knowledge. The reason for this is that since the commitment is perfectly hiding, c is essentially a valid commitment to *every* $s_1 \in \{0,1\}^m$. Thus, every y is "consistent" with c and s_2. Therefore, what we need to ensure is that y is consistent with s_2 and the decommitment of c that are *known* to Party 1. This can be accomplished using a proof of knowledge.

Theorem 9 *Assuming the existence of perfectly-hiding commitment schemes, Protocol 3 is a secure protocol for augmented* perfect *coin-tossing. That is, there exists an efficient transformation of every admissible pair of probabilistic expected polynomial-time machines for the real model (A_1, A_2) into an admissible pair of probabilistic expected polynomial-time machines for the ideal model (B_1, B_2), such that*

$$\{\mathsf{ideal}_{f,\overline{B}}(1^n, 1^n)\} \stackrel{\mathrm{s}}{\equiv} \{\mathsf{real}_{\Pi_2,\overline{A}}(1^n, 1^n)\}$$

where f is the augmented coin-tossing functionality and Π_2 denotes Protocol 3.

The proof of this theorem is very similar to the proof of Theorem 6; the main difference being with respect to the fact that the initial commitment is not perfectly binding.

Acknowledgements. We would like to thank Oded Goldreich for his invaluable contribution to all aspects of this work. We would also like to thank Moni Naor for his suggestion that we look at the question of perfect coin-tossing as well.

References

1. D. Beaver. Foundations of Secure Interactive Computing. In *Crypto91*, Springer-Verlag LNCS Vol. 576, pages 377–391, 1991.
2. D. Beaver, S. Micali and P. Rogaway. The Round Complexity of Secure Protocols. In *22nd STOC*, pages 503–513, 1990.
3. M. Blum. Coin Flipping by Phone. *IEEE Spring COMPCOM*, pages 133–137, February 1982.
4. G. Brassard, C. Crepeau and M. Yung. Constant-round perfect zero-knowledge computationally convincing protocols. In *Theoretical Computer Science*, Vol. 84 (1), pp. 23–52, 1991.

5. R. Canetti. Security and Composition of Multi-party Cryptographic Protocols. *Journal of Cryptology*, Vol. 13, No. 1, pages 143–202, 2000.
6. I. Damgard, T. Pederson and B. Pfitzmann. On the Existence of Statistically Hiding Bit Commitment Schemes and Fail-Stop Signatures. In *Crypto93*, Springer-Verlag LNCS Vol. 773, pages 250–265, 1993.
7. S. Even, O. Goldreich and A. Lempel. A Randomized Protocol for Signing Contracts. *Communications of the ACM* **28**, pp. 637–647, 1985.
8. U. Feige and A. Shamir. Zero-Knowledge Proofs of Knowledge in Two Rounds. In *Crypto89*, Springer-Verlag LNCS Vol. 435, pp 526-544.
9. O. Goldreich. *Secure Multi-Party Computation*. Manuscript. Preliminary version, 1998. Available from: `www.wisdom.weizmann.ac.il/~oded/pp.html`.
10. O. Goldreich. *Foundations of Cryptography: Volume 1 – Basic Tools*. Cambridge University Press, 2001.
11. O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. *SIAM Journal on Computing*, Vol. 25, No. 1, February 1996, pages 169-192.
12. O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *JACM*, Vol. 38, No. 1, pages 691–729, 1991.
13. O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In *19th STOC*, pages 218–229, 1987. For details see [9].
14. S. Micali and P. Rogaway. Secure Computation. Unpublished manuscript, 1992. Preliminary version in *Crypto'91*, Springer-Verlag (LNCS 576), 1991.
15. M. Naor. Bit Commitment using Pseudorandom Generators. *Journal of Cryptology*, Vol. 4, pages 151–158, 1991.
16. M. Naor and M. Yung. Universal One-Way Hash Functions and their Cryptographic Applications. In *21st STOC*, pages 33–43, 1989.
17. A.C. Yao. How to Generate and Exchange Secrets. In *27th FOCS*, pages 162–167, 1986.