Cryptographic Computation: Secure Fault-Tolerant Protocols and the Public-Key Model

(Extended Abstract)

Zvi Galil[1, 2, 3] Stuart Haber[1, 3] Moti Yung[1, 3, 4]

[1] Department of Computer Science, Columbia University
[2] Department of Computer Science, Tel Aviv University

We give a general procedure for designing correct, secure, and fault-tolerant cryptographic protocols for many parties, thus enlarging the domain of tasks that can be performed efficiently by cryptographic means. We model the most general sort of feasible adversarial behavior, and describe fault-recovery procedures that can tolerate it. Our constructions minimize the use of cryptographic resources. By applying the complexity-theoretic approach to knowledge, we are able to measure and control the computational knowledge released to the various users, as well as its temporal availability.

---

[3] Supported in part by NSF grants MCS-8303139 and DCR-8511713.
[4] Supported in part by an IBM graduate fellowship.

C. Pomerance (Ed.): Advances in Cryptology - CRYPTO '87, LNCS 293, pp. 135-155, 1988.

Summary

An important area of research in cryptography is the design of protocols for carrying on certain transactions in a communications network, such as playing poker or holding an election. Many of the protocols proposed in this area have required the expensive on-line generation of a large number of new keys. On the other hand, fundamental research in the traditional problems of cryptography, such as encryption and authentication, has developed the *public-key model*, in which each user has a single validated public key. This model is appropriate to those situations in which generation and validation of new keys is very costly or is otherwise limited. Procedures proposed for this model must preserve the security of the keys. An important question is whether flexible protocol design for a wide variety of problems is possible within the public-key model, so that the expense of generating new keys can be minimized.

We identify a broad class of multi-party *cryptographic computation problems*, including problems subject to the partial-information constraints of the "mental games" of Goldreich, Micali, and Wigderson [24]. In order to solve any cryptographic computation problem, we develop new techniques that can be implemented efficiently in the public-key model. By contrast, the constructions of [42] and [24], which were the first to realize general protocol design in the sense of this paper, require on-line generation of many cryptographic keys. We also consider the the temporal constraint that certain computations occur simultaneously. We formalize this as the problem of multi-party *synchronous computation* and give a solution; it was only recently solved by Yao for the case of two parties [42]. Our tools are *minimum-knowledge* protocols, assuring not only the privacy and synchrony of their computational results but also the security of the users' cryptographic keys.

Much current research has been devoted to fault-tolerance in distributed computation, especially in the cryptographic context. None of the fault-tolerance models presented so far has adequately captured the tradeoff between fault-recovery capabilities and maintenance of security; nor do they describe computation in an unreliable communications environment. We introduce a new fault model that allows a more realistic analysis of faulty behavior. We show how to automatically augment any protocol with procedures for recovery from different kinds of faults. These fault-recovery procedures are implemented in the public-key model, and they enable secure recovery from violation failures in such a way as to preserve the security and privacy of all users, including failing processors.

1. Introduction

Recent work in theoretical cryptography has made great progress toward devising general procedures for solving protocol problems [40, 23, 42, 24]. On the other hand, fundamental research in the traditional problems of cryptography, such as encryption and authentication, has developed the *public-key model*, in which each user has a validated public encryption key and a corresponding private decryption key. In designing a public-key system, the cryptographer must be concerned with the security of encrypted messages, as well as the security of users' keys. For example, recent work on probabilistic encryption schemes [25, 7], as well as the work of Goldwasser, Micali, and Rackoff on "zero-knowledge" interactions [26], were motivated by these security issues. In the public-key model, there is an initialization stage during which each user generates a pair of encryption and decryption keys, announces his public key, and validates it by proving that it was properly chosen. Thereafter, as many cryptographic procedures as possible should be carried out using these keys; the procedures performed by the system must guarantee the security of these keys throughout its lifetime.

It can be quite expensive to generate and validate cryptographic keys. First, if good keys occur sparsely in the set of strings of each length, then generating one at random takes a long time. Second, in certain situations it may not be desirable or possible for every processor in a network to have to generate its own key; for example, some of the processors may be independent robots that are provided with built-in keys. Third, as shown recently by Chor and Rabin [12], the process by which a group of users validate their public keys can be the bottleneck in a multi-party protocol. In light of this, it is inconvenient that recent developments in cryptographic techniques for general design of protocols have made extensive use of on-line generation of encryption keys; over and over again, new keys are generated, used once, and then discarded [13, 23, 42, 24]. While this method allows one to prove the cryptographic security of the proposed procedures, it is wasteful of computational resources. Therefore, for both the theoretical and the practical cryptographer, it is important to know how much can be achieved employing only the users' public keys, and exactly when it is necessary to generate new cryptographic keys. The present work identifies a broad class of multi-party cryptographic problems that can be solved within the public-key model, minimizing the cryptographic resources used in solving them.

Cryptographic protocols are used to solve problems that involve hiding information [37, 25, 14, 5, 20, 19, 40, 24] and forcing certain computations to occur simultaneously [32, 9, 39, 28, 4, 17, 42]. The requirements that individual votes in an election protocol remain private, or that cards drawn from the deck in a poker protocol remain private, are examples of partial-information constraints, while exchange of secrets and signing of contracts are examples of problems subject to temporal constraints. We identify a broad class of multi-party problems called *cryptographic computation problems*. This class includes the "mental games" of Goldreich, Micali, and Wigderson [24], which are computations subject to certain partial-information constraints, as well as problems subject to temporal constraints of synchrony. We define multi-party *synchronous computation* and give a general solution for any computational problem; it was recently solved by Yao for the case of two parties [42].

By a "general solution" to a class of cryptographic computation problems, we mean the following. Given a formal specification of one of these problems, we provide an automatic translation into a multi-party protocol which is correct, and which satisfies the given partial-information and temporal constraints. In order to do this efficiently, we develop new *encryption-based* techniques. Our constructions are

minimum-knowledge protocols that can be implemented efficiently in the public-key model, using public keys that may be based on any family of one-way trapdoor functions. For certain synchronous computation problems, the users may need to cooperate to generate a single additional cryptographic key; for all other cryptographic computation problems, our constructions use only the originally announced public keys of the system. Our minimum-knowledge procedures assure not only the privacy and synchrony of their computational results but also the security of the users' public keys, even after the execution of polynomially many protocols.

If we assume that the Diffie-Hellman key-exchange protocol (based on the discrete logarithm) is secure [16], then we can implement our constructions by simulating the public-key model with only *one* key in the entire network.

Here is an example of the sort of problem for which our techniques can construct a protocol solution (a variation on Yao's millionaires problem [40]): Suppose that a large group of millionaires wish to compute the subset consisting of the one hundred wealthiest among them; the one hundred names that make up this club should only be known to its members and not to the others, and the actual worth of each millionaire should remain secret to everyone. Furthermore, the members of this secret club also wish to generate a common secret encryption key while keeping the anonymity of its owners.

Continuing our study of cryptographic computation, we investigate *fault-tolerance* in the context of cryptographic protocol design. We begin by examining recent work in this area, and observe that none of the models presented so far adequately captures the consequences of users' faulty behavior [13, 23, 24]. The difficulty seems to be that previous models use the notion of faulty behavior that arose in research on the problem of Byzantine agreement. In that domain, a faulty user may attempt to frustrate correct performance by sending different messages to different users, while remaining undetected. In cryptographic protocols, on the other hand, encrypted messages are assumed to be available to all parties. New developments of cryptographic techniques, such as the fact that every NP language has a zero-knowledge interactive proof-system [23], make it possible to design *validated* protocols in which each message is accompanied by an interactive proof that it has indeed been computed correctly. During the execution of a validated protocol, "Byzantine" behavior will be detected (with overwhelming probability) if it is present; thus, it is no longer necessarily the major concern of the protocol designer.

The previous models assumed that all violations, including stop failures, are performed by a malicious adversary; thus the fault-recovery procedures were designed so as to compromise the security of the (presumed) violator --- in fact, to make him give up his identity entirely. This has the paradoxical effect of turning the honest participants into compromisers. One objection to modeling protocols with such a tradeoff between user security and fault-recovery procedures is that these procedures may encourage a real-world adversary, who desires to compromise another user's security, to do this by forcing him to commit a fault (for example by cutting his communication lines, causing a stop-failure). A related objection is that this approach is unsuitable for describing an unreliable communications environment.

We introduce a new fault model for cryptographic protocols. Our model allows a more realistic analysis of faulty behavior, by introducing the cryptographic adversary as a combination of a *compromiser* and a *violator*. We show how to automatically augment any protocol with procedures for

recovery from faults. These fault-recovery procedures make it possible to continue a protocol in the presence of faults as long as there is an honest majority, without changing either the distribution of results computed by the protocol, their privacy, or their synchrony. Furthermore, they enable secure recovery from failures in such a way as to preserve the security and privacy of all users, including failing processors. An important contribution of our new recovery procedures is that they eliminate the security-recovery tradeoff of previous work.

In addition, we present a recovery procedure that enables a violator that stops for a while in the middle of a protocol execution to *rejoin* the protocol later, without disturbing the protocol's computational results or their privacy or synchrony constraints. All of our fault-recovery procedures are implemented in the public-key model.

The multi-party computation problems that we discuss in this paper would all be easy to solve if there were a trusted party among the users. This party could receive each user's inputs (encrypted, if necessary), compute the outputs according to the problem specification, and distribute the outputs (once again encrypted, if necessary) to the appropriate users in such a way as to satisfy the given cryptographic constraints, exactly as desired. Because all the computation would be performed by the ideal trusted party, it would be easy to handle faulty processors. Our goal here, both in our work on techniques for cryptographic computation and in our work on fault-tolerance, is to use the complexity-theoretic approach to knowledge in order to design protocols that achieve the same ends in a distributed fashion, in the absence of such a reliable centralized computer.

To summarize, the combination of the results presented here gives a general technique for solving a large class of distributed problems subject to any combination of partial-information and synchrony constraints, using as few cryptographic resources as possible, and with a secure mechanism for recovery from faults. This suggests what might be called the "trusted-party methodology" of cryptographic protocol design: First design a protocol for the idealized situation in which a single trusted party can easily satisfy all the constraints of the problem, and then replace the trusted party with a cryptographic computation that simulates it as efficiently as possible. The work of [42, 24] can also be viewed in this way.

In the next section, we give the definitions we will use. In section 3 we describe our constructions for solving cryptographic computation problems, and in section 4 we present our new fault model and describe our fault-recovery procedures. Some of the technical details of our constructions are in the appendix.

2. Background and Definitions

2.1. The Public-Key Model

In order to study multi-party protocols, we formalize the interacting parties as a system of *communicating probabilistic Turing machines* [26, 20, 14, 13, 5, 24]. Every machine has a private input tape and a private output tape, a tape of random bits, a working tape, and one public communication tape for each of the other machines. In addition, in order to model the fact that the system is not memory-less,

we assume that each machine has a private history tape on which it records its computation and its communication activities. There is a global clock, and the protocol proceeds in *rounds*; during each round every machine reads its communication tapes, performs some polynomial-time computation, perhaps using some random bits, and then may write messages on the communication tapes. There is also a global parameter giving the size of the computations to be performed and the size of cryptographic keys (i.e., the security parameter). We note that the communication scenario described here is a design model that can be implemented (using an agreement protocol) on more general point-to-point networks.

The system has an initialization stage during which each machine M_i (i=1 . . . n) announces its probabilistic encryption key E_i; it is able to decrypt messages encoded using this key. (This is the setting for the original public-key cryptosystems of [16, 35, 34], which used deterministic encryption functions, and of [25], which introduced probabilistic encryption.) We assume that the keys are based on one-way trapdoor functions drawn from the same distribution, and that these functions are uniformly intractable to invert. In addition, the initialization protocol includes a validation of the keys [12]. We will call this the *public-key model*; the model can be implemented under the general assumption that one-way trapdoor functions exist. Protocols for this model should preserve the security of the cryptographic keys throughout the lifetime of the system.

By an n-party *protocol* we will mean an n-tuple $M=(M_1, \ldots, M_n)$ of such machines. Given the set of keys $E_1, \ldots, E_n$, the global input and its local input, each machine halts in time polynomial in the global parameter with an output string on its output tape, and a string of global output.

For certain applications we extend the model, and specify that the n machines cooperate in order to generate an additional probabilistic encryption key E_0 in such a way that no machine (in fact, no colluding collection of $n-1$ of the machines) knows the corresponding trapdoor information that enables decryption. (In Section 3.2 below we explain how this is done).

2.2. Computational Problems

We are concerned with computational problems as formalized by Yao [42] in the following way. Let $\Sigma=\{0,1\}$. If k is the global parameter, then we will have inputs $(i_0, i_1, \ldots, i_n)$ distributed according to a probability distribution I_k over $(\Sigma^*)^{n+1}$. A problem is specified by requiring that the outputs $(o_0, o_1, \ldots, o_n)$ be distributed according to another probability distribution $O_k(i_0, i_1, \ldots, i_n)$. For example, the special case in which $o_j=f_j(i_0, i_1, \ldots, i_n)$ with probability 1 describes in this framework the problem of computing $n+1$ specified functions of the inputs. Since we are interested in feasible computation, we assume that both the input distributions $\{I_k\}$ and the output distributions $\{O_k\}$ form polynomial-time computable ensembles [41]. Briefly, we will speak of the *computational problem* $[I_k, O_k]$. Without loss of generality (see [33]), the computational problems are given in the form of a probabilistic circuit that computes the desired input-output distribution.

2.3. Protocols and Their Properties

Our aim is to design an n-party protocol $M=(M_1,\ldots,M_n)$ for a computational problem $[I_k,O_k]$, where $E_1,\ldots,E_n$ are the given cryptographic keys. The protocol realizes $[I_k,O_k]$ in a distributed fashion, where i_0 is the public input, i_j is the input of M_j, and similarly o_0 is the public output and o_j is the output computed by M_j. The protocols we design must preserve the security of the keys.

Distributed solutions to a given problem may be subject to cryptographic constraints. For example, we may require that the computation be performed in such a way that certain information remain hidden. Goldreich, Micali, and Wigderson called such problems *partial-information games* (or "mental games") [24]. In particular, we may introduce privacy constraints on inputs, outputs, and intermediate computational results. For example, this may take the form of an anonymity constraint on the identity of the intended recipient of a computed result. Below we formalize these constraints by defining *minimum-knowledge* protocols.

Another constraint we may impose is that of *synchrony*, the requirement that different parties' outputs be computed simultaneously. We remark that the mathematical study of synchrony was initiated in computer science, particularly in the field of distributed computing. Any multi-party computation can be viewed as a competition in which each participant tries to obtain a result significantly earlier than the others; a synchronous protocol is one that prevents anyone from winning. Yao recently proposed a careful definition of the problem of synchronous computation in the case of two parties (calling it the "fairness" constraint), and gave a general solution. Below we give the first solution for the multi-party case.

We define a *cryptographic computation problem* as a composition of one or more computational problems (of the form $[I_k,O_k]$), subject to any combination of partial-information and synchrony constraints. The problem described above in the introduction is one example. Here are two more:

1. Assume that the users of a cryptographic system, each with a private source of randomness, wish to draw public encryption keys at random from a common distribution, one key per user. The parties must somehow combine their private random bits in order to choose their respective keys; however, each trapdoor decryption key chosen should be known only to its owner.

2. Suppose in addition that each of these n users is the representative of a multi-national corporation, and possesses as input data the annual sales and annual income of the company that he or she represents. While desiring to keep their companies' figures secret, the representatives wish to make a certain economic calculation using the data from all n companies. In order to prevent any one representative from computing this output before the others (and then using it in the stock market), they want everybody to learn the result at the same time.

The first problem combines a privacy constraint on the inputs with a privacy constraint on the n-tuple of private decryption keys; the public output is the list of n encryption keys. The second problem is an example of a sychronous computation with input privacy constraints. The problem in the introduction is an example of a computation with an anonymity constraint.

Next, we sketch our definitions that a protocol is correct, and that it satisfies privacy and synchrony

constraints. We will formalize these definitions in the full version of this paper.

The definition of correctness is fairly straightforward. A protocol M is a *correct solution* for the problem $[I_k, O_k]$ if with very high probability its outputs are indeed distributed according to O_k when the input distribution is I_k. It is helpful to think of this as an n-party simulation of the situation in which there is a trusted party that can compute the desired output distribution O_k in a centralized fashion.

During the execution of a protocol M, we will say that a message μ sent by party j is *admissible* if it appears to be appropriate to the specified protocol, i.e. if it seems to be that of the specified machine M_j. (We will formalize this notion of "admissibility" in the full paper.) Temporarily, until we deal with fault-tolerant issues, we assume that a non-admissible message stops the protocol.

Goldreich, Micali, and Wigderson proved that, under the assumption that one-way functions exist, every language in NP has a zero-knowledge protocol for proving membership. This fact has important consequences for the design of protocols [23]. It enables on-line validation that a protocol is being executed correctly. Each message transmission is followed by an interactive proof to all parties that it is indeed a correct message; an incorrect message will be caught with very high probability. This validation can be most efficiently performed by a direct minimum-knowledge simulation of the computation that produced the message, following the construction of [29]. We call such a protocol a *validated* protocol. Our protocols will be validated ones.

A correct validated protocol is one that achieves the computational task that is its goal. Next we deal with controlling the knowledge revealed to the participants in a protocol; we give our definitions that capture those protocols that achieve only the specified task and nothing more.

In order to describe the cryptographic constraints precisely, one might try to give a formal definition of users' changing "knowledge states" during the course of a protocol execution [15]. Instead, we take a computational approach to knowledge and consider certain objects that are theoretically more tractable than knowledge states, namely the sets of strings that may appear on the various tapes of the n Turing machines [26, 20, 42, 23]. We assume that the reader is familiar with the notion of polynomial-time indistinguishable ensembles of strings [25, 41, 26].

In order to define the minimum-knowledge and synchrony properties of a protocol $M=(M_1, \ldots, M_n)$, we must consider the possibility that several of the n parties may *collude* with each other. Colluding parties $j_1, \ldots, j_c$ are able to communicate with each other through private channels --- in particular, they can share their inputs $i_{j_1}, \ldots, i_{j_c}$ and their decryption keys. They may follow any (feasible) programs $M_{j_1}', \ldots, M_{j_c}'$, and the protocol continues as long as the messages they send are admissible. Groups of colluding parties may be created dynamically.

Given a protocol M, consider a set of c colluding parties $j_1, \ldots, j_c$. We desire that they should gain no more information from the messages sent during an execution of the protocol --- e.g. information that can confer a computational advantage in guessing the input or output or compromising the encryption key of another machine that *is* following the protocol --- than they would learn simply by consulting a *result oracle* (which knows *all* the inputs $i_1, \ldots, i_n$ and has access to the colluding machines) to obtain their

output values $o_{j_1}, \ldots, o_{j_c}$. We call a protocol *minimum-knowledge with respect to c(n)-subsets* if it satisfies this requirement with respect to every possible dynamically chosen subset containing as many as $c(n)$ of the machines. Of course, the strongest possible requirement is that $c(n) = n-1$.

We formalize the definition of the minimum-knowledge property by stipulating that for any subset of possibly colluding machines, there should exist a probabilistic polynomial-time *simulator S*, as follows. Given access to the internal state and all the tapes of the colluders, and allowed queries to the result oracle for the colluding machines' outputs, S produces a simulation of the colluders' *views* of an actual execution of the protocol. (Each machine's view, recorded on its history tape, includes all messages sent during the execution and the random-tape bits it has used in its own computation, as well as any privately computed outputs.) The simulation should be indistinguishable, by any feasible computation that may use the colluders as subroutines, from a transcript of the colluders' views of an actual execution. This definition generalizes the notions of "minimum-knowledge" and "result-indistinguishable" protocols that transfer computational knowledge [26, 20] to the case of n-party distributed computation.

It is a consequence of this definition that if a protocol M is minimum-knowledge, then the security (in the sense of [25]) of the key E_j of any non-colluding machine M_j is not compromised by the protocol.

As in the case of two-party minimum-knowledge protocols [20], it is crucial to our proofs of the theorems in this paper that if two multi-party protocols are both minimum-knowledge, then so is their concatenation. We will give a proof of this concatenation lemma in the full paper.

Next we define synchrony for a protocol M. Once again, consider a set of c colluding parties. We desire that at any moment the colluders can gain no more computational advantage over any other machine in computing the outputs than they already have simply by knowing their inputs. A protocol will be called *synchronous with respect to c(n)-subsets* if it satisfies this requirement with respect to every subset containing as many as $c(n)$ of the machines.

More precisely, at any fixed time during the protocol execution, consider the problem of distinguishing which of two given strings (drawn from the space of possible computed outputs defined according to *a priori* knowledge of the inputs) will turn out to be the actual result of the computation. The protocol achieves synchrony if the colluders' best distinguishing probability between two different strings in their result-space is essentially the same as the best distinguishing probability for any non-colluding user.

3. General Protocol Design

In this section we present a general technique for constructing a protocol solution to any cryptographic computation problem in the public-key model. Given a circuit for the computational problem $[I_k, O_k]$ and a set of cryptographic constraints that the solution must satisfy, we construct a protocol solution. The technical tool we introduce is an *encryption-based* circuit simulation, motivated by the recent work of Yao [42] and Goldreich, Micali, and Wigderson [24] on protocol design, and by the security provided by probabilistic cryptosystems [25, 41, 8, 7].

We state here the main result of this section.

Theorem 1: Given a set of polynomially many n-party cryptographic computational problems, there is a correct polynomial-size minimum-knowledge protocol solution. The protocol uses only the public keys $E_1,...,E_n$, if there are no synchrony constraints among the given computational problems.

Our proof is a constructive one, automatically translating the given problems to a solution protocol. Below we present our techniques.

3.1. Two-Party Computation

First we consider the case of two parties, A and B. We are given a circuit that has output distribution $O_k(i_A,i_B)$ when its input (i_A,i_B) is distributed according to I_k. For simplicity, suppose that O_k is the desired output for B. We describe here a minimum-knowledge two-party protocol that simulates the given circuit. We contrast our solution for the public-key model with the protocol solution of Yao [42], which required the on-line generation of $O(C)$ cryptographic keys for a circuit of size C.

Our construction uses probabilistic encryption and cryptographically secure pseudo-random bit generators based on the users' public keys [6, 41, 31]. (See also [30, 10, 8, 2, 7, 38, 21].) If we make the special assumption that the Diffie-Hellman key-exchange protocol (based on the discrete logarithm) is secure [16], then we can also implement all of our constructions by simulating the public-key model with only *one* key in the entire network.

The protocol has several stages. During the first stage one of the parties, the *circuit-constructor* A, prepares an encryption-based circuit-simulation for the use of the other party, the *circuit-evaluator* B. B then verifies that A's construction is as specified in the computation problem. Next, A and B combine their inputs in such a way that they remain secret, and finally B evaluates the output.

A uses his probabilistic encryption key to construct the circuit-simulation, and uses a minimum-knowledge proof to validate the construction. The simulation protocol proceeds by simulating the gates of the given circuit. The protocol consists of three kinds of (simulated) *gates*: input gates, intermediate gates and output gates. Each gate has *entries* for input and output, and a "truth table" that maps the values of the input entries to the output entries. An entry, representing a bit in the computation, consists of two random bit-string *cleartexts*, one for each possible value of the bit; the entry is presented to B as the pair of encoded *ciphertexts* of the two cleartexts. A *valued* entry is one whose cleartexts have low-order bit (for example) equal to the bit represented. An entry may be either *ordered*, in which case the two cleartexts represent 0, 1 (in that order), or *unordered*, in which case they represent either 0, 1 or 1, 0 (at random). An entry may also be *marked*: each of the ciphertexts is tagged with a probabilistic encryption of the bit that it represents.

The inputs to the simulation are the input entries of the input gates. A's input entries, corresponding to the bits of i_A, are unordered and marked, while those of B, corresponding to the bits of i_B, are ordered and unmarked. The truth table of every gate consists of "rows", each of which enables the evaluator to combine one cleartext of each of the gate's input entries in order to obtain the cleartext of an output entry; the rows of the truth table are permuted at random. The output of each input or intermediate gate "connects" to the input of the next gate in the circuit, because the first gate's output entry enables B to

decrypt one of the input-entry ciphertexts of the next gate. These intermediate entries of the circuit are unmarked and unordered. The output gates of the circuit have output entries that are valued.

B uses the circuit-simulation in three stages: an input stage, a computation stage, and an output stage. During the input stage *B* gets his input --- i.e., receives from *A* the cleartext corresponding to each of his input bits --- by the *generalized oblivious transfer* protocol [24]. (This is an extension of the *oblivious transfer* [27, 19] that can be implemented in the public-key model. It provably assures that *B* receives just one of the two cleartexts, according to his choice, while *A* does not learn which one is transferred). In case an encryption of the input bit is publicized by *B* in advance, *B* also interactively proves that he has received the correct cleartext. Then *A* does the following for each of his (unordered) input entries: without revealing its tag, he sends *B* the cleartext that represents the value of his input bit. (Thus *B* does not learn what this value is.) In case an encryption of the input bit is known, *A* also proves to *B* that he sent the correct cleartext.

Once he has a cleartext for each of a gate's input entries, *B* computes a cleartext for that gate's output entry according to the truth table. (We sketch the details of the truth table in appendix I.) Since this output entry is not ordered, *B* cannot tell what bit-value this cleartext represents. Using the "connections" between gates, *B* then continues through the circuit, computing output cleartexts for the intermediate gates and, finally, for the output gates of the circuit. Since these, the output entries of the circuit, are valued, *B* learns the values of the output bits, i.e. the result of the required computation. If desired --- i.e. if O_k is the desired output for *A* as well as for *B* --- *B* can now tell the result to *A*, and convince him of the value of each output bit by revealing the corresponding cleartext he computed. The entire interaction is minimum-knowledge: neither *A* nor *B* learns more about the other's inputs, or about the others' decryption key, than they would if an oracle simply told them the output. (More formally, in the actual computation each party "learns" only random values; given the output by an oracle, each of them could generate a simulated protocol transcript that is polynomial-time indistinguishable from an actual one.)

Further technical details of our construction are described in appendix I.

3.2. Multi-Party Computation

Now we consider the general case of n parties. Given a circuit for an n-party computational problem, the above construction for two parties can be used as a basic step in order to give an n-party minimum-knowledge protocol that simulates the circuit. We contrast our solution for the public-key model with the protocol solution of [24], which required the on-line generation of $O(n^2C)$ cryptographic keys for a circuit of size C.

At any point during the course of the circuit-simulation, each bit b in the simulated computation is shared by the n parties: each party holds a secret share of the global bit, but this share is random and gives no information about the global value. Each gate of the simulated circuit is replaced by an n-party computation whose inputs are the n-tuples of shares of the gate's inputs, and whose outputs form a similar distribution of n shares of the gate's output. This n-party computation consists of $n(n-1)$ two-party interactions as above.

The construction of [24] uses Barrington's encoding of circuits with elements of the permutation group

S_5 [3]. Each user's share of a globally distributed bit is a permutation, and two-party circuits of Yao [42] are used in order to "compute" with these permutations. In joint work with Silvio Micali, we simplify the representation and manipulation of distributed shares. The simplification is that each user's share of b is simply a random bit b_j satisfying $b = \sum_{j=1}^{n} b_j \pmod 2$. We show how to compute with these shares, by making some simple bit-computations, in appendix II.

Continuing the work described above, several authors have recently proposed protocols for different sorts of circuit simulation, basing their cryptographic security on certain complexity-theoretic assumptions [22, 1, 11].

3.3. Synchronous Computations

Given a circuit C for an n-party computational problem, we give a protocol solution that satisfies the synchrony constraint.

Yao gave a protocol for synchronous two-party computation [42]. As in the construction of section 3.1 (which was motivated by his work), the two parties have asymmetric roles in his protocol; one party "constructs" the circuit simulation, while the other party "evaluates" it. In order that the computation be synchronous, Yao proposes the following: first, the two parties together generate a trapdoor function whose secret trapdoor they do not know; next, they use this function to encrypt their computational results; and finally, they cooperate to recover the trapdoor simultaneously.

In order to implement multi-party synchronous computation, the n parties jointly generate a random cryptographic key E_0 that will be used in order to hide the computational results, in such a way that all n parties must cooperate in order to recover them. At any point in the recovery process, all parties have equal computational advantage in computing the desired result. We describe the details of our solution in the full paper.

4. Fault-Tolerance

When we introduce faults to our model of computation we complicate the protocol-designer's task: users may not follow their instructions, they may try to extract additional information from the messages sent during the execution, or they may completely stop functioning. Up to this point, we have assumed that all parties behave admissibly, and otherwise the protocol stops; thus the correct completion of the protocols we have described depends on this assumption. Where it is possible, we would like to augment our protocols with the capability to detect and recover from faulty behavior. Recent work [13, 23, 24] has dealt with this problem. In this section we give a careful analysis of faulty behavior in the context of cryptographic protocols, and present new protocols procedures for recovery from faults.

An important tool for fault-tolerant protocol design is that of *verifiable secret-sharing* [13, 36]. A verifiable secret-sharing protocol with parameters (m,t) is a procedure by which a sender can distribute to each of m receivers a *share* of a given secret s, in such a way that it is easy to verify that each share is indeed a proper share, it is infeasible to obtain any information about s from any l of the shares, as long as $l<t$, and it is easy to compute s from any t of the shares. A protocol that achieves this can be given (for

any $t \leq n$).

Using verifiable secret sharing, Goldreich, Micali, and Wigderson devised a procedure that transforms any given protocol into a validated protocol tolerating up to $t(n)$ faults, where $t=\lfloor (n-1)/2 \rfloor$. Faulty processors may deviate from their specified algorithms in any arbitrary but feasibly computable way. Here we sketch their transformation. Let $h=n-t$. The transformed protocol begins with a set-up phase during which each processor $(n-1,h)$-verifiably secret-shares its input and its random tape. Until this phase is over, the protocol is vulnerable to faults; the only way to continue the protocol is by restarting the set-up procedure. When a fault is detected during the execution of the transformed protocol, the faulty processor loses its identity completely: the honest users use their secret-shares to reconstruct its input and random tape, and then simulate its role throughout the rest of the protocol. The transformation is *correctness-preserving*, in that honest parties compute the same outputs as in the original protocol; and it is *privacy-preserving*, in that whatever a $t(n)$-bounded adversary can compute in the transformed protocol, he could also have computed in the original one [23, 24]. If the original protocol is minimum-knowledge with respect to c-subsets, for any $c<h$, then so is the transformed protocol.

Notice that in the procedure just outlined, a faulty user is always severely punished; its input is compromised, even if its violation was an unintentional stop-failure caused, for example, by an undelivered message. The procedure is based on the pessimistic assumption that all faults are malicious ones, which only holds in a fully reliable communications environment. The notion of faulty behavior implicit in this assumption coincides with that of research in Byzantine agreement. In that domain faulty behavior is undetected, and messages are private. In cryptographic protocols, on the other hand, encrypted messages are assumed to be available to all parties. Furthermore, during the execution of a validated cryptographic protocol "Byzantine" behavior is detected (with overwhelming probability) whenever it occurs. Since all players know this is true, in practice it may be the case that most faults will be unintentional. Paradoxically, the transformation of Goldreich, Micali, and Wigderson turns the honest majority into compromisers. Here we present a more realistic analysis of possible faulty behaviors, and describe a new fault-recovery procedure that avoids imposing on inadvertently faulty processors the heavy penalty demanded by their procedure.

4.1. New Fault Model

The first question one must ask is when it is possible for a protocol to recover even from a single fault. If the protocol is to continue after a violation, then any recovery procedure must confer on the honest users that detect the faulty processor the power to act on its behalf in the continuation of the protocol. But there are protocols --- e.g. poker-playing, stock transactions --- in which we don't want to recover in this way; even if a user (perhaps inadvertently) times out of today's transactions, he may want to play cards (or buy stocks) tomorrow, using the same secret card-playing (or stock-market) strategy. This strategy is a part of the processor's program that should never be shared with others. It is only in the case of protocols in which each processor's entire program is publicly known that it makes sense to look for recovery procedures. For the rest of this paper, we assume that all protocols are of this form.

We begin by distinguishing between two different sorts of faulty behavior: passive *compromise*, by which we mean the attempt to extract secret information while seeming to act according to the protocol,

and active *violation*, intentional or unintentional, of the protocol's instructions, such as by sending erroneous (Byzantine) messages or by stop-failure. We model these two kinds of faults by speaking of an adversary who is able to corrupt a bounded number of the processors. The adversary may, at any time during the execution of our protocol, choose a processor to compromise, until he has chosen $c(n)$ *compromisers*; he has access to the internal state and all the tapes of each of the compromisers, and he may perform any (feasible) computation with this information. Compromising behavior is, by definition, undetectable by an honest user; the most that we may demand is that the protocol be designed so that compromisers cannot gain anything from the information that they share. In fact, a protocol that is minimum-knowledge with respect to $c(n)$-subsets can tolerate a $c(n)$-bounded compromising adversary. The adversary may also choose up to $v(n)$ *violators*; at any point during the execution, he may block the messages sent by any of the violators or over-write them with messages of his choice. From the point of view of an honest processor, violating behavior is of two sorts: stop-failure, which is detectable when the violator times out (according to the global clock) and fails to send an expected message during a round of the protocol; and sending erroneous messages, which will be detected (with very high probability) if our protocol is a validated one. It can happen that a processor can be both a compromiser and a violator; this situation can model what the authors of [24] call a "malicious adversary", which is the strongest form of adversarial behavior. However, we do not assume that every fault must be malicious.

To fix notation, let $h(n) = n - c(n) - v(n)$ denote the minimum number of *honest* processors, those that are neither compromisers nor violators.

4.2. Fault-Recovery Procedures

We suggest a new protocol transformation that fits the more realistic fault model just described. The transformed protocol preserves the privacy of *all* inputs, including those of a violator, and can be implemented in the public-key model. More precisely, the transformation is *discretion-preserving*: during an execution of the transformed protocol an honest user computes no more about the input of any other user, including violators, than he computes in the original protocol. Furthermore, we give a *rejoin procedure* by which a disqualified faulty machine may rejoin the protocol execution in order to obtain its computed outputs, without disturbing the original protocol's computational results or the cryptographic constraints they satisfy.

Here we state the main result of this section.

> **Theorem 2:** Any n-party protocol may be transformed into a validated protocol that is secure against $c(n)$ compromisers and tolerates $v(n)$ (disqualified and rejoining) violators, as long as $v(n) + c(n) \leq n-1$; the transformation is of polynomial cost, and is correctness-preserving, privacy-preserving, and discretion-preserving.

The transformed protocol begins with a *set-up phase*, during which processors' inputs are distributed (by verifiable secret-sharing); in order to qualify for participation in the protocol a machine must take part in this stage. When a violation occurs --- up to $v(n)$ of them are possible --- the set-up procedure restarts. When a violation is detected during the *execution phase*, the violating machine is *disqualified* and the *recovery procedure* is invoked.

Let $M = (M_1, \ldots, M_n)$ be the original protocol. The transformed protocol $M' = (M_1', \ldots, M_n')$ proceeds as follows.

SET-UP PHASE

Each machine M_i', for each bit b of its given input and its generated random tape, chooses bits b_j ($j=1 \ldots n$) at random satisfying $b = \oplus_{j=1}^{n} b_j$, and sends $E_j(b_j)$ to M_j'. (We will call b_j a *piece* of the *distributed* bit b.)

Each machine M_i' follows a verifiable secret-sharing protocol with parameters $(n-1, h)$ to share with all the other machines its piece of every distributed bit.

EXECUTION PHASE

Follow M, validating every message. Whenever M requires a disqualified machine M_j to compute a message μ, the active machines simulate the computation of μ, using the distributed bits of M_j (via a multi-party cryptographic computation protocol as described in Section 3.2).

Upon detection of a violation by machine i, invoke the protocol RECOVER(i).

When disqualified machine M_i' requests to rejoin the execution, invoke the protocol REJOIN(i).

RECOVER(i)

Disqualify machine M_i'. Let j be the first index in $\{i+1 \bmod n, i+2 \bmod n, \ldots\}$ such that machine M_j' is not disqualified.

For each globally distributed bit b, the active machines reconstruct the violator's piece b_i, and machine M_j' updates its piece by setting $b_j := b_j \oplus b_i$)

Note that it is possible to recover in this way, even if another processor should fail during a reconstruction step.

REJOIN(i)

For each globally distributed bit b, and for each machine M_j' that is not disqualified, M_i' chooses a bit r_j at random, and performs a 2-party simulated computation with M_j' that has the effect of setting $b_j := b_j \oplus r_j$; then M_i' sets $b_i := \oplus_j (r_j)$.

Observe that the number of times that the REJOIN procedure may be invoked is bounded by the length of the original protocol, since at least one step must intervene between a disqualification and a rejoining.

We call attention to a trade-off between the maximum number of compromisers and the maximum number of violators that can be handled using these methods. By the minimum-knowledge properties of verifiable secret-sharing, the violators' privacy is preserved as long as $c(n) \le h(n)-1$. Since h users are needed in order to reconstruct each violator's shares, the maximum number of violators that can be tolerated satisfies $v(n) \le n-h(n) \le n-c(n)-1$; in other words, we must have $v(n)+c(n) \le n-1$. In case there is no distinction made between compromisers and violators, the maximum number of faults is $\lfloor (n-1)/2 \rfloor$.

In the full version of this paper, we will describe a *dynamic recovery* procedure for a model in which there is a failure rate (instead of bounds $c(n)$ and $v(n)$) given as a fault parameter.

5. Conclusions

In this paper we address a number of issues in the design of protocols to solve general multi-party computational problems subject to various cryptographic constraints. First, we present new techniques that enable the automatic translation of a problem specification into a multi-party protocol satisfying any given partial-information and temporal constraints; the resulting protocol can be implemented in the public-key model, requiring the generation of new cryptographic keys only for certain synchronous computation problems. Second, we present new fault-recovery procedures that make it possible to continue a protocol in the presence of faults as long as there is an honest majority, without changing either the distribution of results computed by the protocol or the cryptographic constraints they satisfy, while at the same time preserving the security and privacy of all users, including failing processors.

In both parts of this work we apply the complexity-theoretic approach to knowledge in order to measure and control the computational knowledge released to each of the participants in a protocol. This approach enables us to design protocol procedures that simulate, in a distributed fashion, a central trusted party that assures the correctness, privacy, and synchrony of the results computed. This "trusted-party methodology" of protocol design can be applied, for example, to existing protocols that require a central server (e.g. [14, 18]) in order to make them fully distributed.

Appendix

I. The Basic Two-Party Construction

Here we describe the technical details of a simulated gate in a two-party circuit simulation.

For this description, we implement the operations of probabilistic encryption as follows. The one-way function E is used to encode the cleartext a by choosing a bit-string r at random and computing the ciphertext $x = E(r) \oplus a$. We will call r the *random seed* for the ciphertext x. Without the trapdoor information for E, it is an intractable problem to recover a (or any value that functionally depends on a) from x.

As in Section 3.1, we call the two parties A and B (for Alice and Bob). A is the circuit-constructor and B is the circuit-evaluator.

Instead of giving our construction in complete generality, we will explain the case of an input gate with two input entries that computes $i_A \,\mathrm{OR}\, i_B$, where i_A is A's input bit and i_B is B's input bit. The two entries are presented to B as two pairs of ciphertexts, $[x_0, x_1]$ and $[y_0, y_1]$, unordered and ordered, respectively.

Next, A and B perform the generalized oblivious transfer protocol, so that (according to his choice of the bit i_B=0, say) B obliviously receives the random seed s_0 for the ciphertext y_0; he can then decode it and recover the cleartext b_0. A then will send B either r_0, the random seed for x_0, or r_1, the random seed for x_1, depending on whether A's input bit i_A is 0 or 1. Say, B receives r_1; he can decipher x_1 and recover the cleartext a_1.

Note that after these computations it is an intractable problem for B to extract a_0 from x_0, or b_1 from y_1.

Let a_0^L, a_0^R denote the left and right halves of the bit-string a_0, and similarly for the other cleartext strings. Along with its entry ciphertexts, the gate is presented with "instructions" of the following form.

- 1st, 1st : L, L
- 1st, 2nd : R, L
- 2nd, 1st : L, R
- 2nd, 2nd : R, R

What this means is that if the two cleartexts received by B correspond to the first and the first (respectively) of the two pairs of entry ciphertexts, then B should compute the XOR of the left half and the left half (respectively) of the two cleartexts; if they correspond to the first and the second, he should XOR the right half of the first with the left half of the second; and so on. Each instruction corresponds to a line of the gate's truth table, and the lines have been randomly permuted.

The cleartexts have been chosen at random by A so as to satisfy the equations

- $a_0^L \oplus b_0^L = c_0$,
- $a_0^R \oplus b_1^L = c_1$,
- $a_1^L \oplus b_0^R = c_1$, and
- $a_1^R \oplus b_1^R = c_1$,

where c_0 and c_1 are random strings of the appropriate length. (Explicitly, A could build the gate by first choosing the order of the four instructions, then choosing c_0, c_1, a_0, a_1 at random, and finally computing b_0, b_1 according to the equations.)

In our case, B has received a_1 and b_0, so he follows the third instruction and computes c_1. The fact that he also knows the string b_0^L gives him no help in using the first equation to compute c_0, since he has no information about the random string a_0^L. Similarly, since he does not know which of the other three "rows" of the truth table would give him the same result-string c_1 as the row which he did use, he cannot use the string a_1^R to learn anything from the fourth equation. And he knows neither of the bit-strings on the left hand side of the second equation.

Thus, in any execution of the evaluation protocol, B computes a single result-string, and gains no computational advantage in guessing any other result-string; and this is what we need in order to assure the security of intermediate computational results. The result string is used in turn as a random seed, either for an input entry of the intermediate gate to which this gate is connected, or for the simulated circuit's output.

II. Multi-Party Computation

Here we show how n parties can use their shares of globally distributed bits to simulate circuit computations.

Throughout this section, all arithmetic expressions should be interpreted mod 2. At the beginning of the protocol, each machine does the following for each bit b that it needs as input to the circuit simulation: it chooses bits b_j ($j=1 \ldots n$) at random satisfying $b = \sum_{j=1}^{n} b_j$ and sends $E_j(b_j)$ to M_j.

How do n parties "compute" with these shares? Suppose a and b are two shared bits; that is, machine M_i possesses share a_i of bit a and share b_i of bit b. These random shares satisfy the equations $a = \sum_{i=1}^{n} a_i$ and $b = \sum_{i=1}^{n} b_i$.

To simulate a NOT gate, i.e. to simulate the computation $a := 1-a$, one of the machines, say the first, complements its share: $a_1 := 1-a_1$. Now every machine possesses a proper random share of the complementary bit $1-a$. For the simulation of an AND gate, i.e. of the computation $c := ab$, consider the following equations, where each r_{ij} is a bit randomly chosen by M_i:

$$ab = (\sum_{i=1}^{n} a_i)(\sum_{i=1}^{n} b_i) = \sum_{1 \le i,j \le n} a_i b_j = \sum_{i=1}^{n} a_i b_i + \sum_{i \ne j} [r_{ij} + (r_{ji} + a_i b_j)].$$

This sum can be rewritten as

$$\sum_{i=1}^{n} \{ a_i b_i + \sum_{j \ne i} [r_{ij} + (r_{ji} + a_j b_i)] \} = \sum_{i=1}^{n} c_i ,$$

where $c_i = a_i b_i + \sum_{j \ne i} [r_{ij} + (r_{ji} + a_j b_i)]$ is the share of c that will be computed by M_i at the end of the computation. Each pair of machines, M_i and M_j, carry on two different two-party interactions. In the first of these, M_i acts as the circuit-constructor and M_j acts as the circuit evaluator for a circuit with inputs r_{ij}, a_i from M_i and b_j from M_j, and output $r_{ij} + a_i b_j$ for M_j. Machine M_i holds the value r_{ij} and machine M_j holds the value $r_{ij} + a_i b_j$; thus the two machines hold random shares whose sum is equal to $a_i b_j$. In the second of their interactions, they reverse their roles in order to share $a_j b_i$. (Notice that the required two-party computations are very simple.) Each machine M_i takes the sum mod 2 of these $2(n-1)$ output values, and adds $a_i b_i$ to it; the resulting sum c_i is its share of the new globally distributed bit $c=ab$. This concludes the simulation of the AND gate.

The multi-party circuit-simulation just described is minimum-knowledge with respect to $(n-1)$-subsets, and it can realize computations with privacy constraints. At the end of the simulation, the circuit's outputs are "computed" as follows. For each bit b of the public output o_0, each machine M_i announces its share b_i. For each bit b of the private output o_j for machine M_j, every other machine M_i, $i \ne j$, reveals its share b_i.

Acknowledgements

We would like to thank Silvio Micali for introducing us to this area of research and for many helpful discussions, and Cynthia Dwork for her careful critical remarks, as well as the members of the Dipartimento di Informatica ed Applicazioni at the University of Salerno, who were our hosts when we began the research reported here.

References

1. Abadi M., and J. Feigenbaum. A Simple Protocol for Secure Circuit Evaluation. Preprint, 1987.

2. Alexi, W., Chor, B., Goldreich O. and Schnorr C.P. RSA/Rabin Bits are 1/2 + (1/poly(k)) Secure. Proc. 25th FOCS, IEEE, 1984, pp. 449-457.

3. Barrington, D.A. Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC_1. 18th STOC, ACM, May, 1986, pp. 1-5.

4. Ben Or, M., O. Goldreich, S. Micali, and R. Rivest. A Fair Protocol for Signing Contracts. Proceedings of ICALP-85, July, 1985, pp. 43-52.

5. Benaloh, J.C. and Yung M. Distributing the Power of a Government to Enhance the Privacy of Voters. Proc. 5th PODC, ACM, 1986, pp. 52-62.

6. Blum, M. and Micali, S. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. Proc. 23rd FOCS, IEEE, 1982, pp. 112-117. Also in: *SIAM Journal on Computing*, November 1984, 850-864.

7. Blum, M. and S. Goldwasser. An Efficient Probabilistic Public-Key Scheme Which Hides All Partial Information. Proceedings of Crypto84, 1985, pp. 289-301.

8. Blum, L., Blum M. and Shub M. Comparison of Two Pseudo-Random Number Generators. Proceedings of Crypto82, August, 1982, pp. 61-78.

9. Blum, M. "How to Exchange (Secret) Keys". *ACM Transactions on Computer Systems 1*, 2 (May 1983), 175-193.

10. Boppana, R.B. and R. Hirschfeld. Pseudorandom Generators and Complexity Classes. Preprint, 1986.

11. Chaum D., I. Damgard, and J. van de Graaf. Multiparty Computations Ensuring Secrecy of Each Party's Input and Correctness of the Output. These proceedings.

12. Chor, B. and Rabin M.O. Achieving Independence in Logarithmic Number of Rounds. 6th PODC, ACM, August, 1987.

13. Chor, B., Goldwasser S., Micali S. and Awerbuch B. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. Proc. 26th FOCS, IEEE, 1985, pp. 383-395.

14. Cohen, J.C. (Benaloh) and Fischer M.J. A Robust and Verifiable Cryptographically Secure Election Scheme. Proc. 26th FOCS, IEEE, 1985, pp. 372-383.

15. DeMillo, R.A., N. Lynch and M. Merritt. Cryptographic Protocols. 14th STOC, ACM-SIGACT, May, 1982, pp. 383-400.

16. Diffie, W., and Hellman M.E. "New Directions in Cryptography". *IEEE Transactions of Information Theory IT-22* (November 1976), 644-654.

17. Even, S., Goldreich O. and Lempel A. "A Randomized Protocol for Signing Contracts". *Communications of the ACM 28*, 6 (June 1985), 637-647.

18. Feige, U., A. Fiat and A. Shamir. Zero-Knowledge Proofs of Identity. 19th STOC, 1986, pp. 210-217.

19. Fischer, M., S. Micali, C. Rackoff, and D. Wittenberg. An Oblivious Transfer Protocol Equivalent to Factoring. Manuscript, 1986.

20. Galil, Z., Haber S. and Yung M. A Private Interactive Test of a Boolean Predicate and Minimum-Knowledge Public-Key Cryptosystems. Proc. 26th FOCS, IEEE, 1985, pp. 360-371.

21. Goldreich, O., S. Goldwasser, and S. Micali. How to Construct Random Functions. Proc. 25th FOCS, IEEE, 1984, pp. 464-479.

22. Goldreich O., and R. Vainish. How to Solve Any Protocol Problem: an Efficiency Improvement. These proceedings.

23. Goldreich, O., S. Micali and A. Wigderson. Proofs that Yield Nothing But their Validity and a Methodology of Cryptogrphic Protocol Design. 27th FOCS, IEEE, October, 1986, pp. 174-187.

24. Goldreich, O., S. Micali and A. Wigderson. How to Play Any Mental Game. 19th STOC, 1987, pp. 218-229.

25. Goldwasser, S. and Micali S. Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information. Proceedings of the 14th Annual ACM Symp. on Theory of Computing, ACM-SIGACT, May, 1982, pp. 365-377.

26. Goldwasser, S., S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof-Systems. 17 STOC, ACM-SIGACT, May, 1985, pp. 291-304.

27. Halpern, J. and Rabin M.O. A Logic to Reason about Likehood. Proc. 15th STOC, ACM, 1983, pp. 310-319.

28. Hastad, J. and A. Shamir. The Cryptographic Security of Truncated Linearly Related Variables. 17th STOC, ACM-SIGACT, May, 1985, pp. 356-362.

29. Impagliazzo R., and M. Yung. Direct Minimum-Knowledge Computations. These proceedings.

30. Kranakis, E.. *Primality and Cryptography*. John Wiley and sons, Chichester, 1986.

31. Levin, L. One-way Functions and Pseudorandom Generators. Proc. 17th STOC, ACM, 1985.

32. Luby, M. , Micali S. and Rackoff C. How to Simultaneously Exchange a Secret Bit by Flipping a Symmetrically-Biased Coin. 24 FOCS, IEEE, November, 1983, pp. 11-22.

33. Pippenger, N., and M.J. Fischer. "Relations among Complexity Measures". *Journal of the ACM 26* (1979), 361-381.

34. Rabin. M. O. Digitalized Signatures and Public-key Functions as Intractable as Factorization. LCS/TR-212, MIT, January", 1979.

35. Rivest, R. , Shamir A., Adleman L. "A Method for Obtaining Digital Signatures and Public Key Cryptosystems". *Communications of the ACM 21*, 2 (February 1978), 120-126.

36. Shamir, A. "How to Share a Secret". *Communications of the ACM 22*, 11 (November 1979), 612-613.

37. Shamir, A., Rivest R. Adleman L. Mental Poker. In *Mathematical Gardner*, Klarner D. E., Ed., Wadsworth Intrntl, 1981, pp. 37-43.

38. Vazirani, U. and Vazirani V. Efficient and Secure Pseudo-Random Number Generation. Proc. 25th FOCS, IEEE, 1984, pp. 458-463.

39. Vazirani, U. and Vazirani V. Trapdoor Pseudo-random Number Generators, with Applications to Protocol Design. 24th FOCS, IEEE, November, 1983, pp. 23-30.

40. Yao, A. Protocols for Secure Computations. 23rd FOCS, IEEE, November, 1982, pp. 160-164.

41. Yao, A. Theory and Applications of Trapdoor Functions. 23rd FOCS, IEEE, November, 1982, pp. 80-91.

42. Yao, A. How to Generate and Exchange Secrets. 27th FOCS, IEEE, October, 1986, pp. 162-167.