J. Cryptology (1991) 4: 75–122

Journal of Cryptology

Secure Multiparty Protocols and Zero-Knowledge Proof Systems Tolerating a Faulty Minority[1]

Donald Beaver
AT&T Bell Laboratories, 600 Mountain Avenue,
Murray Hill, NJ 07974, U.S.A.

Abstract. A multiparty protocol to compute a function $f(x_1, \ldots, x_n)$ operates as follows: each of n processors holds an input x_i, and jointly they must compute and reveal $f(x_1, \ldots, x_n)$ without revealing any additional information about the inputs. The processors are connected by secure communication lines but some number of processors may be corrupted by a resource-unbounded adversary that may attempt to interfere with the protocol or to gain extra information. Ben-Or, Goldwasser, Wigderson, Chaum, Crépeau, and Damgård have given protocols tolerating faults in $t < n/3$ processors. We improve the bound to $t < n/2$; as long as a majority remains uncorrupted, general and secure computations are achievable. To address and prove the security of our results, we introduce concise definitions for security and fault-tolerance. In particular, our notion of *relative resilience*—a means to compare the security and fault-tolerance of one protocol with that of another in a formal manner—provides a key tool for understanding and proving protocol security.

Key words. Distributed computing, Fault tolerance, Secret sharing, Zero knowledge, Proof systems.

1. Introduction

Inspired by the growth of large distributed systems, the problem of achieving secure and reliable computation in a network of processors has received a great deal of recent attention [20], [21], [14], [15], [2], [29], [3], [10], [17], [27], [4].

Each processor in the network holds some private information, denoted x_i, and together the processors must compute some general function $f(x_1, \ldots, x_n)$, without revealing the individual inputs. For example, each input x_i might represent an election ballot cast by the owner of a workstation; the network of processors must calculate the tally without revealing the votes. Unfortunately, no single processor is completely reliable, either in terms of mechanical failure or incorruptibility by an adversary. Hence it is impossible to arrange for some central, trusted processor to collect the inputs and return the function value.

[1] Date received: January 31, 1990. Date revised: November 1, 1990. This research was supported in part under NSF Grant CCR-870-4513. This work was done while the author was a graduate student at Harvard University.

The processors must therefore arrange to compute the result without a trusted party. To their advantage, a synchronous, completely connected communication network, with secure channels and broadcast channels, is available. Other sorts of networks can be considered—most notably, *broadcast* networks, in which each message is available for all processors to read—but solutions in these scenarios usually require that an adversary be computationally bounded and that unproven complexity theoretic assumptions be made. This work considers information-theoretic security, in which no computational bounds on the power of attacking parties are required, and no unproven assumptions are made.

The contributions of this paper are twofold. On the one hand, this work presents new definitions for security that provide a concise way not only to understand but to prove security. On the other hand, this work develops algorithmic techniques to solve the problem of *secure multiparty computation* and to solve a related problem, that of *zero-knowledge proofs in a distributed network* without unproven assumptions.

A common approach to describing intuitively the meaning of security in multiparty protocols for general function computation uses the notion that the secrecy ensured by a trusted party should be the standard by which a general protocol is measured. That is, the execution of a protocol should reveal nothing more than is revealed in the situation where a trusted party is available, namely, nothing more than the value of $f(x_1, \ldots, x_n)$. Other properties, such as correctness, independence of input selection, and fairness are equally important, and the literature contains a variety of approaches to describing and defining these desired properties.

One of the primary contributions of this work is a simple definition for *resilience*, the combination of *security* and *reliability*. This definition is based on comparing the resilience of a general protocol with that of an *ideal* protocol, in which a trusted processor is in fact present. To compare one protocol with another, a notion of *relative resilience* is necessary. Relative resilience allows not only the comparison of a given protocol to an ideal situation but the comparison of any pair of protocols, which in turn supports modular proofs of security. A protocol may be proven as secure as some intermediate protocol, which is then proven as secure as an ideal protocol. The composition of protocols may also be proven as secure as an ideal protocol that computes the composition of the functions each individual protocol computes.

Our approach is more general and provides a simpler approach than notions of *fault-oracles* proposed first as a definition of privacy by Galil *et al.* [18] and later extended to cover correctness as well, independently by Kilian *et al.* [30], [12], [26] and by Beaver [5]. Goldwasser and Levin [22] and Beaver and Goldwasser [10] employ a definition stating that any execution of a "robust" protocol must correspond to a *possible* execution of the same protocol without an adversary (and such that the inputs of uncorrupted players are the same in both cases). All of these notions seek, with varying degrees of clarity, to connect a general protocol to an ideal, uncorrupted computation. All of these notions use a fixed comparison of a general protocol to a fixed ideal. The definitions presented in this paper make these ideas explicit, and provide a much broader and more robust means to compare *any* pair of protocols.

The direct comparison of a protocol to an ideal situation is only one application

of our general definition. In fact, the general notion of relative resilience provides simple definitions for security and fault-tolerance for a diverse set of problems in interactive computation (e.g., zero-knowledge proof systems, Byzantine Agreement, oblivious transfer, two-party oblivious circuit evaluation, etc.).

With a clear set of definitions for security, the next step is to show that resilient protocols are possible. In fact, a variety of (unproven) positive results have been achieved. In the so-called *cryptographic* model, in which all processors, including any adversary, are polynomially time bounded and certain unproven assumptions are made, Goldreich *et al.* [20], [21] pioneered methods which allowed faults in $t < n/2$ processors in the network (here, t is an upper bound on the number of faults). Improvements in technique and efficiency were given [18], [25], notably by Galil *et al.* [18]. Beaver and Goldwasser [10] pushed the bounds of fault-tolerance above the threshold of $n/2$, with the restriction that either all processors learn the correct result in a fair fashion, or the nonfaulty processors can identify and eliminate corrupted processors.

When no complexity-theoretic assumptions are made, the task of achieving high fault-tolerance is difficult. Achieving a resilience of $n/2$ or more is impossible in general [14], [17], [27], [4]. Ben-Or *et al.* [14] and Chaum *et al.* [15] used Shamir's techniques for secret sharing [31] in an elegant way to prove that secure multiparty protocols are possible for $t < n/3$. Secret sharing is a technique whereby information is distributed robustly among the members of a network such that no sufficiently small collection of members determines anything about the information. (In Shamir's method, a dealer, holding a secret s from some field E, creates a random polynomial $p(u)$ of degree t such that $p(0) = s$, and sends $\text{PIECE}_i(s) = p(i)$ to each player i. Any t to fewer pieces are uniformly distributed, preserving secrecy, while any $t + 1$ unaltered pieces determine s.) Essentially, the protocols of [14] and [15] consist of several intermediate steps to create new secrets whose values are sums or products of earlier secrets, without revealing any of the secrets. With *absolute* security, no bound better than $t < n/3$ is achievable, but in the range $n/3 \leq t < n/2$, it has remained unknown whether secure protocols were achievable with some negligible chance of error.

The methods of [14] and [15] for $t < n/3$ do not work for the case of $t < n/2$ for two reasons. The first is that Shamir's methods for secret sharing are not robust against such large numbers of faults. A method for Verifiable Secret Sharing (VSS) [16], in which the network checks that a secret is shared properly before using it in a computation, is required. The second reason is that, even given VSS, methods for creating new secrets from old secrets are not robust. In particular, an intermediate step, in which one participant demonstrates that three secrets a, b, and c satisfy $c = ab$ without revealing their values, is necessary for adaptations of [14] and [15] to work. Call this problem the *ABC Problem*. The works of [14] and [15] solve the ABC Problem for $t < n/3$ but not for $t < n/2$.

Solving the first problem, Rabin [28], [29] provides an elegant method to achieve VSS when $t < n/2$. The method includes additional information in the form of "check vectors" to ensure that values are not changed.

This paper presents a solution to the second problem, by developing the first technique for the ABC Problem when $t < n/2$, using VSS as a subroutine. Resilient multiparty protocols for $t < n/2$ fall out as an immediate consequence.

A second consequence of the solution to the ABC problem presented here is that it is possible for one member of a network to prove in zero-knowledge [23] that the value of a given secret y is the result of some function p applied to secrets $x_1, \ldots, x_m$, without revealing the secret values themselves. This proof technique generalizes the ABC Problem, works with exponentially small probability of failure, and requires no unproven complexity theoretic assumptions, in contrast with the two-party case, where such proof systems are currently impossible without unproven assumptions. The communication complexity of secure zero-knowledge network proof systems is not large: the number of rounds depends on the size of the network (or is constant for $t < n/3$; see [2]), but not on the complexity of the function, p; and the number of message bits sent is polynomial in the size of a circuit for p. The latter measure, message size, can be reduced to a polynomial in the size of the network, independent of the circuit complexity of p, using techniques of Beaver and Feigenbaum [7] and Beaver *et al.* [8].

A different solution to the ABC Problem was independently developed by Rabin and Ben-Or [29], but that solution requires bit-by-bit arithmetical operations. The solution presented here uses arithmetic over a finite field, saving many rounds of interaction. A large number of rounds of interaction is usually the greatest hindrance to practical applications.

Neither definitions of security nor proofs of security for the methods of VSS in [28] and [29] have yet appeared at the time this goes to press, and certainly no proofs of security using the definitions presented here have appeared. For mathematical soundness, however, all protocols and subprotocols must be proven resilient. Rather than attempt to prove [28] and [29] correct—even though the definitions of this paper provide a solid foundation for such a task—an alternate method for VSS that developed out of work with Feigenbaum and Shoup [9] is presented and proven correct. Using a set of lemmas regarding protocol composition and related issues, the first proof of resilience for a multiparty protocol is presented.

This paper is organized as follows. In Section 2 notation, details of protocol execution with adversaries, general and ideal protocols, and induced distributions are described. In Section 3 new and concise definitions for resilience and presented and discussed. In Section 4 general methods for proving security and presented along with proofs of some previously unsupported folk theorems. In Section 5.1 new methods for verifiable secret sharing are presented. In Section 5, and Section 5.4 in particular, the main result is presented: a multiparty protocol resilient against faults in up to half the network. In Section 6 zero-knowledge proof systems in the presence of a network are proven possible. The philosophical and definitional heart of this work (relative resilience) lies in Section 3, and the algorithmic heart of this work (solving the ABC Problem for $t < n/2$) lies in Section 5.4.

2. Preliminaries

2.1. *Notation*

The vector notation describes a labeled set of items: $\vec{x} = \{(1, x_1), \ldots, (n, x_n)\}$. A subscript denotes a subset of those values: $\vec{x}_T = \{x_i | i \in T\}$; $[n]$ denotes $\{1, 2, \ldots, n\}$.

Labels are ordinarily omitted for readability: $\vec{x} = \{x_1, \ldots, x_n\}$. Let E be a finite field, usually taken to be GF(2^n) or GF(p) for some prime p, $n < p \leq 2n$. Fix an alphabet $\Sigma = \{0, 1\}$; extensions to Σ (e.g., the delimiters #, @, $\langle$, $\rangle$, " , ", and the special symbol Λ signifying absence of a message (as opposed to an empty message)) are encoded naturally. If $S = \{s_1, \ldots, s_n\}$ is a set of strings in lexicographical order, then S is encoded as $\langle s_1, s_2, \ldots, s_n \rangle$. If S' is a set of objects, each of which has a natural encoding as a string, then the encoding of S' is the same as the encoding of the set S of encodings of each of its members. A vector is encoded as a set, and sometimes $\vec{x}$ is used to represent the string encoding of the vector.

The notation "i:" in a protocol description indicates the local computations and variable assignments of player i. The notation "$i \rightarrow j$: m" indicates that i sends m to j, and "$i \rightarrow [n]$: m" means that i broadcasts m. The notation "$(1 \leq i \leq n)$" indicates that the succeeding text is performed in parallel for i in the range from 1 to n.

The probability of event Y with respect to distribution P is denoted $\Pr_P[Y]$. Let dist(X) denote the set of all distributions on a set X. A *probabilistic function* is a function whose range is a set of distributions, namely $f: X \rightarrow \mathsf{dist}(Y)$ for some Y. Though the "output" of such a function is a distribution, it is sometimes convenient to refer to an output as a sample from that distribution.

The difference of two distributions P and Q on a set X is defined by $|P - Q| = \sum_{x \in X} |\Pr_P[x] - \Pr_Q[x]|$. A sample taken according to P is denoted by $x \leftarrow P$. A sample taken according to P subject to the condition that $A(x)$ holds is denoted by $\{x \leftarrow P | A(x)\}$. If $f: X \rightarrow \mathsf{dist}(Y)$ and $g: Y \rightarrow \mathsf{dist}(Z)$, then $g \circ f: X \rightarrow \mathsf{dist}(Z)$ is defined by

$$\Pr_{g \circ f(x)}[z] = \sum_{y \in Y} \Pr_{f(x)}[y] \cdot \Pr_{g(y)}[z].$$

"Deterministic" functions (i.e., functions) may also be included. The composition of a function h with a probabilistic function f is described by the following notation, intended to suggest an "experiment" to produce a sample from the resulting distribution:

$$h \circ f(x) = \{y \leftarrow f(x): h(y)\}.$$

The uniform distribution on a set X is denoted *uniform*(X). The uniform distribution on the set $\mathsf{Poly}(t, s)$ of polynomials $f(u)$ of degree t satisfying $f(0) = s$ is denoted $UPoly(t, s)$. Fix a finite field E and consider a fixed set of points $\alpha_1, \ldots, \alpha_n$, normally either $\{1, \ldots, n\}$ or $\{1, \omega, \ldots, \omega^{n-1}\}$ where ω is a primitive nth root of unity. The distribution on sets of n values obtained by selecting a polynomial at random according to $UPoly(t, s)$ and evaluating it at n points is defined by

$$\mathsf{Pieces}(n, t, s) = \{\vec{y} \in E^n | (\exists f \in \mathsf{Poly}(t, s))(\forall i) y_i = f(\alpha_i)\},$$

$$UPieces(n, t, s) = \mathit{uniform}(\mathsf{Pieces}(n, t, s)).$$

Shamir's method for secret sharing takes a sample $\vec{y} \leftarrow UPieces(n, t, s)$ and sends y_i to player i.

An *ensemble* is a particular kind of probabilistic function, namely a family $\mathscr{P} = \{P(z, k)\}$ of distributions on $\Sigma^{\leq p(|z|, k)}$, parametrized by $z \in \Sigma^*$ and $k \in \mathbf{N}$, where $p(\cdot, \cdot)$ is polynomially bounded.

Definition 1. Ensembles $\mathscr{P}$ and $\mathscr{Q}$ are $\delta(k)$-*indistinguishable*, written $\mathscr{P} \approx^{\delta(k)} \mathscr{Q}$, if

$$(\forall k) \quad (\forall z) \qquad |\mathscr{P}(z, k) - \mathscr{Q}(z, k)| \leq \delta(k).$$

Under the standard O-notation, $f(k) = O(g(k))$ means $(\exists c, k_0)$ $(k \geq k_0 \Rightarrow f(k) \leq c \cdot g(k))$. The value k_0 is called the *convergence parameter.*

Definition 2. Ensembles $\mathscr{P}$ and $\mathscr{Q}$ are $O(\delta(k))$-*indistinguishable* if

$$(\exists \Delta: \mathbf{N} \to \mathbf{N}) \qquad \mathscr{P} \approx^{\Delta(k)} \mathscr{Q} \quad \text{and} \quad \Delta(k) = O(\delta(k)).$$

Perfect (i.e., 0) indistinguishability is written $\mathscr{P} \approx \mathscr{Q}$. Exponential (i.e., $O(c^k)$ for some constant $0 \leq c < 1$) indistinguishability is written $\mathscr{P} \approx^{e} \mathscr{Q}$. Statistical (i.e., $O(k^{-c})$ for all constants $c > 0$) indistinguishability is written $\mathscr{P} \approx^{s} \mathscr{Q}$. The notion of computational indistinguishability concerns the ability of a polynomial-time Turing machine to distinguish the two ensembles, but this is of no concern in this paper.

The class of probabilistic finite functions PFF computed by protocols mapping n-vectors of m-bit arguments to distributions on n-vectors of m-bit arguments is defined as

$$\text{PFF} = \{F | F: \Sigma^* \to \text{dist}(\Sigma^*), F((\Sigma^m)^n) \subseteq \text{dist}((\Sigma^m)^n)\},$$

with the restriction that, for each value of n and m, the restriction of F to $(\Sigma^m)^n$ is computed by a probabilistic circuit $C_{n,m}$. A circuit is a directed, acyclic graph whose nodes are labeled AND, OR, and NOT (arithmetic versions over a finite field are also possible), evaluated in a natural way. Input nodes are those with no incoming edges and output nodes those with no outgoing edges. A probabilistic circuit has some number $r(m, n)$ of distinguished input nodes; the distribution computed by such a circuit is induced naturally by assigning these nodes uniformly random values from $\{0, 1\}$. Without loss of generality, functions that output a single m-bit string or even a single bit may be considered.

2.2. *Protocol Execution*

Informally, a *protocol* is a collection of sets of Turing machines, one set of n machines for every $n \in \mathbf{N}$. Each Turing machine has some number of input and output communication tapes, one for each communication channel; without loss of generality we take each machine M to have a single input–output tape on which incoming messages are written in lexicographic order, tagged with an integer stating the identity of the communication line. At the start, the values 1^n, 1^m, 1^k, x, and a are written on the input tape: n represents the number of machines in the network; m, the number of bits in the input; k, a security parameter; x, the input of that machine; and a, an auxiliary input of that machine. Each machine also has a work tape W and a "random" tape R, namely a particular tape on which a "random" sequence of bits is placed. A "random" tape is a convenient intuitive representation for the idea that each machine represents not a deterministic function (on input tape $I = n\#m\#k\#x\#a$ and "random" tape R), but a *probabilistic* function (on $n\#m\#k\#x\#a$) defined as follows. Let $\varphi(R)$ denote the rightmost location

M reaches on R, let H be the set of R's on which M halts, and let $\psi = \{\mathbf{pref}_{\varphi(R)}(R) | R \in H\}$, where $\mathbf{pref}_a(b)$ denotes the first a letters of b. Let $\psi(z) = \{R \in \psi | M(n\#m\#k\#x\#a, R) = z\}$. Then M computes the probabilistic function δ defined by

$$\Pr_{\delta(n\#m\#k\#x\#a)}[z] = \frac{\sum_{R \in \psi(z)} 2^{-|R|}}{\sum_{R \in \psi} 2^{-|R|}}.$$

Each Turing machine enters a waiting state when its computation is complete, and is restarted later with a new set of incoming messages; the work tape is left unchanged. Without loss of generality, each machine can accept a description $v(r)$ of all previous inputs for some number r of activations and can compute the next output based on $v(r)$.

The only reasons to adopt a Turing machine model are for concreteness and to supply a measure of communication and computational complexity. A more general definition is the following.

Definition 3. A *player* is a tuple $(Q, q(0), \delta, Y)$, where Q is a (possibly infinite) set of states, $q(0) \in Q$ is an initial state,

$$\delta: Q \times 2^{\Sigma^*} \to \mathrm{dist}(Q \times 2^{\Sigma^*})$$

is a probabilistic transition function, and $Y: Q \to \Sigma^*$ is called the *output function.*

The state of a Turing machine player is, under this definition, a superstate combining its tape contents and its finite control; the transition function arises naturally from the transition function of the Turing machine; and the output function maps the superstate to a string describing the contents of the machine's work or output tape (or to Λ if the output tape has an infinite word written on it). The advantages of using a higher level of generality are twofold. Only the input/output behavior of the players is important; the rest can be abstracted away, and the analysis of security need only consider the given probabilistic functions. Computations need not be recursive functions, thus allowing an adversary to "gain" tremendous power by corrupting a potentially "tremendously strong" player; protocols resilient against stronger adversaries provide greater assurance of security. Note that the protocols presented here do require only polynomial-time Turing machine computations; weak players can participate, but adversaries of unbounded computational power are permitted.

A channel describes how a message sent by one player is transformed into the message received by another player or by several other players.

Definition 4. A *channel* is a probabilistic function:

$$C: \Sigma^* \to \mathrm{dist}(2^{\mathbf{N} \times \mathbf{N} \times \Sigma^*}).$$

The probability that a given sequence $((i, j_1, \mu_1), \ldots, (i, j_l, \mu_l))$ is output by channel C when player i applies message μ is $\Pr_{C(\mu)}[\{(i, j_1, \mu_1), \ldots, (i, j_l, \mu_l)\}]$.

Typical channels include the *private channel* between i and j, for which $\Pr_{C(\mu)}[(i, j, \mu)] = 1$, and the *broadcast channel* for player i, for which $\Pr_{C(\mu)}[\{(i, 1, \mu), (i, 2, \mu), \ldots, (i, n, \mu)\}] = 1$. Another common channel is the noisy (Oblivious Transfer) channel from i to j, for which $\Pr_{C(\mu)}[\{(i, j, (1, \mu))\}] = \frac{1}{2}$ and $\Pr_{C(\mu)}[\{(i, j, (0, 0))\}] = \frac{1}{2}$.

A synchronous, n-player, R-round protocol is executed as follows. Each of n players begins with an input x_i and an auxiliary input a_i on its input tape. Before the protocol begins, player i is in state $q(i, 0)$. Round 0 consists of an initial computation, $(q(i, 1), \mu^{\text{out}}(i, [n], 1)) \leftarrow \delta_i(q(i, 0), x_i \# a_i)$. (In general, $q(i, r + 1)$ represents the state of player i after its computation in round r, and $\mu^{\text{out}}(i, [n], r + 1)$ the messages it intends to send. The notation $\mu(A, B, r)$ represents a string of messages from players in set A to those in set B.) Channel functions are applied to the undelivered messages, and the outputs, #-delimited and lexicographically ordered, are placed on the input tapes of the recipients. The messages to be delivered to player i are denoted $\mu^{\text{del}}([n], i, r + 1)$.

In each round, each player computes locally on the input messages and its current state, producing new output messages (ignored in the final round) and a new state. In the case of *Turing machine* players, the "state" represents a superstate consisting of the finite control state and work-tape contents, and the local computation is a recursive transduction from these and the random-tape contents to new tape contents, finite control state, and outgoing messages. A Turing machine that fails to halt or that gives an output that cannot be parsed is considered to output an empty message string, $\varnothing$.

Definition 5. A *protocol* Π is a family of sets of players along with (labeled) set of channels: $\Pi = \{(P_1, \ldots, P_n, \mathscr{C}_n) | n \in \mathbf{N}\}$, where $\mathscr{C}_n = \{(c_1, C_1), \ldots, (c_{N(n)}, C_{N(n)})\}$ for some function N, and $c_i \in \Sigma^*$ is a label for the channel C_i: $\mathbf{N} \times \Sigma^* \to \text{dist}(2^{\mathbf{N} \times \mathbf{N} \times \Sigma^*})$. Each player P_i accepts an input of the form $1^n \# 1^m \# 1^k \# x_i \# a_i$, where $|x_i| = m$. We consider only protocols in which the computations of each uncorrupted player are independent of its auxiliary input.

It is sometimes convenient to refer to $\{P_1, \ldots, P_n\}$, for a given value of n, as a protocol. A *universal* protocol includes a label of a function to compute along with the inputs, normally in the form of a string description of a circuit.

For purposes of formal security analysis, it is best to abstract away mechanical details—which often vary according to taste—and consider the execution of a protocol simply as a composition of probabilistic functions. The *algorithmic* specification of an execution described in Fig. 1 can be represented as a probabilistic function mapping $\langle \vec{x}, \vec{a} \rangle$ to a string of final states and views of the execution, $\langle \vec{q}(R), \vec{v}(R) \rangle$, by composing the following probabilistic functions. Here, each state is described by a string; the superstate of a Turing machine (including tape contents) is a concatenation of a description of its finite control and current state, along with the contents of all tape locations over which the head has passed.

- Lexicographically order a set of strings:

$$\textbf{Ord}(s_1 \# \cdots \# s_N) = s_{i(1)} \# \cdots \# s_{i(N)},$$

where $i(1) \cdots i(N)$ is a permutation such that $(\forall j)\ s_{i(j)} \leq s_{i(j+1)}$.

Execute Protocol

1 $(1 \le i \le n)$ $\mu^{\text{in}}([n], i, 0) \leftarrow 1^n \# 1^m \# 1^k \# x_i \# a_i$.
For $r = 0..R(n, m, k) - 1$ **do begin**
r.1 $(1 \le i \le n)$ i: Compute locally:
$(\vec{q}(i, r+1), \mu^{\text{out}}(i, [n], r+1)) \leftarrow$
$\delta_i(\vec{q}(i, r), \mu^{\text{in}}([n], i, r))$.
r.2 Apply channel functions to $\mu^{\text{out}}([n], i, r+1)$ to obtain $\mu^{\text{del}}([n], [n], r+1)$.
r.3 Sort messages to obtain $\mu^{\text{in}}([n], [n], r+1)$.
r.4 Update views.
end
R(n,m,k).1 Final computation after protocol:
$(\vec{q}(i, R(n, m, k)+1), \mu^{\text{out}}(i, [n], R(n, m, k)+1)) \leftarrow$
$\delta_i(\vec{q}(i, R(n, m, k)), \mu^{\text{in}}([n], i, R(n, m, k)))$
R(n, m, k).2 $(1 \le i \le n)$ i: **output** $\vec{q}(i, R(n, m, k)+1)$

Fig. 1. Description of steps involved in executing a protocol on $(n, m, k, \vec{x}, \vec{a})$. Messages μ^{out} report what the processors wish to send; μ^{del} represents messages output from the channels, which become incoming messages μ^{in} for the next round. With adversaries, μ^{del} may be altered before it becomes μ^{in}.

- String manipulation:

$$\langle A_1, \ldots, A_N \rangle \odot \langle B_1, \ldots, B_N \rangle = \langle\langle A_1, B_1 \rangle, \ldots, \langle A_N, B_N \rangle\rangle,$$
$$\odot^{-1}(\langle\langle A_1, B_1 \rangle, \ldots, \langle A_N, B_N \rangle\rangle) = \langle\langle A_1, \ldots, A_N \rangle, \langle B_1, \ldots, B_N \rangle\rangle.$$

- Convert standard and auxiliary inputs to an initial global state vector:

$$\textbf{Init}(n \# m \# \vec{x} \cdot \vec{a}, k) = \langle\langle q_1(0), \ldots, q_n(0) \rangle,$$
$$\langle 1^n \# 1^m \# 1^k \# x_1 \# a_1, \ldots, 1^n \# 1^m \# 1^k \# x_n \# a_n \rangle,$$
$$\langle \varnothing, \ldots, \varnothing \rangle\rangle.$$

- Perform all local computations:

$$\delta(\langle q(1), \ldots, q(n) \rangle, \langle \mu(1), \ldots, \mu(n) \rangle) = \odot^{-1}(\langle \delta_1(q(1), \mu(1)), \ldots, \delta_n(q(n), \mu(n)) \rangle),$$
$$\textbf{Local}(\langle \vec{q}, \mu^{\text{in}}, \vec{v} \rangle) = \langle \vec{q}, \mu^{\text{in}}, \vec{v}, \delta(\vec{q}, \mu^{\text{in}}) \rangle.$$

- Evaluate channel functions:

$$\textbf{Send}(\langle C_1, m_1 \rangle \# \cdots \# \langle C_N, m_N \rangle) = C_1(m_1) \# \cdots \# C_N(m_N).$$

- Extract messages for player j from a list:

$$\textbf{Select}(j, \langle i_1, j_1, m_1 \rangle \# \cdots \# \langle (i_N, j_N, m_N) \rangle)$$
$$= \textbf{Ord}(\langle (i_{k(1)}, j_{k(1)}, m_{k(1)}) \rangle \# \cdots \# \langle (i_{k(l)}, j_{k(l)}, m_{k(l)}) \rangle),$$

where $\{k(1), \ldots, k(l)\} = \{k | j = j_k\}$.

- Sort a list of messages according to recipient:

$$\textbf{Sort}(L) = \langle \textbf{Select}(1, L), \ldots, \textbf{Select}(n, L) \rangle,$$

where n is the maximal recipient number.

- Pass messages through channels:

$$\textbf{Channel}(\langle \vec{q}, \mu^{\text{in}}, \vec{v}, \langle \vec{q}^{\,\text{new}}, \mu^{\text{out}} \rangle\rangle) = \langle \vec{q}, \mu^{\text{in}}, \vec{v}, \vec{q}^{\,\text{new}}, \mu^{\text{out}}, \textbf{Send}(\mu^{\text{out}}) \rangle.$$

- Produce a new global state vector describing the entire system, and update views:

$$\textbf{Update}(\langle \vec{q}, \mu^{\text{in}}, v, \vec{q}^{\,\text{new}}, \mu^{\text{out}}, \mu^{\text{del}} \rangle)$$
$$= \langle \vec{q}, \textbf{Sort}(\mu^{\text{del}}), \vec{v} \odot \mu^{\text{in}} \odot \vec{q}^{\,\text{new}} \odot \mu^{\text{out}} \odot \textbf{Sort}(\mu^{\text{del}}) \rangle.$$

- Perform one round of local computation and message delivery:

$$\textbf{Round} = \textbf{Update} \circ \textbf{Channel} \circ \textbf{Local}.$$

- Extract outputs from final states:

$$\textbf{Outputs}(\vec{q}) = \langle Y_1(q_1), \ldots, Y_n(q_n) \rangle.$$

- Convert global state vector to output and view:

$$\textbf{Out}(\langle \vec{q}, \mu, \vec{v} \rangle) = \textbf{Outputs}(\vec{q}) \cdot \vec{v}.$$

Definition 6. An R-round *execution* of a synchronous n-party protocol on m-bit inputs is the following ensemble:

$$\textbf{Exec}(n \# m \# \vec{x} \cdot \vec{a}, k) = n \# m \# \textbf{Out}(\textbf{Round}^{R(n,m,k)}(\textbf{Init}(n \# m \# \vec{x} \cdot \vec{a}, k))).$$

The n and m values are often omitted for clarity. In particular, **Exec** maps an input string of the form $\vec{x} \cdot \vec{a}$ to an output string of the form $\vec{y} \cdot \vec{v}$.

A slightly more general definition might operate the protocol until all uncorrupted players enter a special, *finished* state. Such a modification is easily incorporated.

The execution of a protocol in the presence of an adversary takes a slightly different course. The adversary examines a subset of the messages to be delivered and substitutes its own. The coalition of players whose messages are read and altered may be chosen statically or dynamically. A passive adversary changes no messages; a Byzantine adversary is permitted to do so. A rushing adversary may examine and replace messages from newly corrupted players before the entire set of messages is delivered.

Definition 7. A *t-adversary* is a pair $(\mathcal{A}, T)$ where $\mathcal{A}$ is a player with one communication line, with state set Q_A, transition function δ_A, and output function Y_A. Here, T: $Q_A \to [n]$ is a *coalition function*, namely a function describing the set of players corrupted so far by $\mathcal{A}$; T is "nondecreasing" and never exceeds t corruptions: $(\forall s \in \Sigma^*)\ T(q_A) \subseteq T(\delta_A(q_A, s))$ and $(\forall q_A)\ |T(q_A)| \leq t$.

This definition requires the adversary to remember, for the sake of our convenience, which processors it has corrupted. The communication line is used to transmit requests for corruptions, responses containing information from those corruptions, and messages to replace those from corrupted players.

For full generality we describe protocol execution with a dynamic, rushing, Byzantine adversary. The description applies equally well to passive or static adversaries, even though it specifies that the adversary request corruptions and replace outgoing messages; a passive adversary, for example, "replaces" messages

from faulty players with the identical, correct message (e.g., it keeps an internal copy of players it has corrupted and continues to operate them *exactly* as specified by the protocol, thereby effecting no change).

The execution of a protocol in the presence of an adversary is a modification of the adversaryless execution. In particular, all nonfaulty players perform local computations and generate messages that are then passed through the communication channels. At this point, the adversary takes action. It requests incoming messages—from both the previous round and those generated in the current round, rushed—and outgoing messages, view, and state for a particular player by generating a request $i \in [n]$. Letting $Q = \langle \vec{q}, \mu^{\text{in}}, v, \vec{q}^{\text{new}}, \mu^{\text{out}}, \mu^{\text{del}} \rangle$, define

$$\textbf{Fault}(i, Q) = \langle q_i, \mu^{\text{del}}([n], i), v_i, \mu^{\text{out}}(i, [n]) \rangle,$$

$$\textbf{Request}(Q@\langle q_A, i \rangle) = \begin{cases} Q@\langle q_A, i \rangle, & i \notin [n], \\ Q@\delta(q_A, \textbf{Fault}(i, Q)), & i \in [n]. \end{cases}$$

After up to n rounds of learning new information (t for t-adversaries, who restrain themselves to requesting only t corruptions), $\mathscr{A}$ produces a set of messages to replace those from faulty players; these messages are passed through the channels. Views and states of corrupted players are nullified.

$$\textbf{Replace}''(i, T, v, w) = \begin{cases} v, & i \in T, \\ w, & i \notin T, \end{cases}$$

$$\textbf{Replace}'(T, \vec{v}, \vec{w}) = \langle \textbf{Replace}''(1, T, v(1), w(1)), \ldots, \textbf{Replace}''(n, T, v(n), w(n)) \rangle,$$

$$\textbf{Replace}(Q@\langle q_A, \mu \rangle) = \langle \vec{q}, \mu^{\text{in}}, \textbf{Replace}'(T(q_A), \vec{v}, \vec{0}), \textbf{Replace}'(T(q_A), \vec{q}^{\text{new}}, \vec{0}), \mu^{\text{out}}, \textbf{Replace}'(T(q_A), \mu^{\text{del}}, \textbf{Send}(\mu)) \rangle @ q_A.$$

The corruption state during a given round is concisely described by the following probabilistic function:

$$\textbf{Corrupt} = \textbf{Replace} \circ \textbf{Request}''.$$

Note that this expression is very general; it allows any number of corruptions, and does not specify any restrictions on the adversary. When a particular *adversary class* is described (for example, t-adversaries, polynomial-time adversaries, etc.), it is the specification of the adversary class that "restrains" the adversaries' behavior (e.g., to no more than t corruptions), not the description of protocol execution with an adversary.

Extend **Init** and **Out** so that

$$\textbf{Init}(n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k) = \textbf{Init}(n \# m \# \vec{x} \cdot \vec{a}, k) @ \delta_A(q_A(0), 1^n \# 1^n \# 1^k \# a_A),$$

$$\textbf{Out}(\langle \vec{q}, \mu, \vec{v} \rangle @ q_A) = \textbf{Outputs}(\vec{q}) \circ \vec{v} \circ Y_A(q_A).$$

The n and m are often omitted for clarity. Extend the other functions f defined previously for adversaryless n-player protocol execution so that $f(Q@q_A) = f(Q)@q_A$. Define

$$\textbf{RoundA} = \textbf{Update} \circ \textbf{Corrupt} \circ \textbf{Channel} \circ \textbf{Local}.$$

Definition 8. An R-round *execution* of an n-party protocol on m-bit inputs in the presence of an adversary $\mathscr{A}$ is the following ensemble:

$$\mathbf{ExecA}(n\#m\#\vec{x}\cdot\vec{a}\cdot a_A, k) = n\#m\#\mathbf{Out}(\mathbf{RoundA}^{R(n,m,k)}(\mathbf{Init}(n\#m\#\vec{x}\cdot\vec{a}\cdot a_A, k))).$$

The n and m values are often omitted for clarity. In particular, **ExecA** maps an input string of the form $\vec{x}\cdot\vec{a}\cdot a_A$ to an output string of the form $\vec{y}\cdot\vec{v}\cdot y_A$. To specify a particular protocol Π and adversary $\mathscr{A}$, we write $\mathbf{ExecA}[\Pi, \mathscr{A}](n\#m\#\vec{x}\cdot\vec{a}\cdot a_A, k)$.

As before, the definition may be modified to allow the number R of rounds to depend on the number of repetitions of **RoundA** before all nonfaulty players reach a quiescent, *finished* state. The *round complexity* of a protocol is the maximal number of such repetitions; the *message complexity* is the maximal net size of all messages from nonfaulty players; and the *local complexity* is the maximal amount of time required by a Turing machine player to compute outgoing messages from incoming messages, namely to evaluate δ_i. The last is distinguished from the *computational complexity* of the function the protocol purports to compute. Computation of a particular function is discussed later.

The outputs of the players and of the adversary are often of greater significance than the views of the players. We define

$$Y(\vec{y}\cdot\vec{v}\cdot y_A) = \langle y_A, y_1, \ldots, y_n\rangle,$$
$$Y_A(\vec{y}\cdot\vec{v}\cdot y_A) = \langle y_A\rangle,$$
$$Y_{[n]}(\vec{y}\cdot\vec{v}\cdot y_A) = \langle y_1, \ldots, y_n\rangle.$$

Definition 9. The *ensemble of outputs* $[\Pi, \mathscr{A}]$ induced by protocol Π with adversary $\mathscr{A}$ includes outputs of the players and the adversary:

$$[\Pi, \mathscr{A}](n\#m\#\vec{x}\cdot\vec{a}\cdot a_A, k) = Y(\mathbf{ExecA}(n\#m\#\vec{x}\cdot\vec{a}\cdot a_A, k)).$$

Relevant portions are the ensembles describing adversary outputs and player outputs:

$$[\Pi, \mathscr{A}]^{Y_A}(n\#m\#\vec{x}\cdot\vec{a}\cdot a_A, k) = Y_A(\mathbf{ExecA}(n\#m\#\vec{x}\cdot\vec{a}\cdot a_A, k)),$$
$$[\Pi, \mathscr{A}]^{Y_{[n]}}(n\#m\#\vec{x}\cdot\vec{a}\cdot a_A, k) = Y_{[n]}(\mathbf{ExecA}(n\#m\#\vec{x}\cdot\vec{a}\cdot a_A, k)).$$

2.3. *Real and Ideal Protocols*

An *Adversary class* is a set of adversaries. Unless otherwise specified, we consider dynamic, rushing, Byzantine adversaries. The *t-adversary class* A^t consists of t-adversaries whose coalitions are arbitrary t-subsets of $[n]$ (see Definition 7). A *real protocol* is one associated with a t-adversary class; in particular, no player is above corruption.

In contrast, an *ideal protocol* contains one or more incorruptible players, called *trusted hosts*. These players are numbered $n+1$, $n+2$, etc., for convenience. The *ideal t-adversary class* $\mathsf{A}^t_{\text{ideal}}$ contains adversaries designed against ideal protocols but, as before, allowed only to corrupt t-subsets of $[n]$.

An ideal protocol represents a situation that is too risky to assume in general. In

ID(F)

1. $(1 \le i \le n)$ $i \to n+1$: x_i

2.1 $(1 \le i \le n)$ $n+1$: $x_i' = \begin{cases} x_i, & \text{if } x_i \in \text{dom}(F), \\ 0, & \text{otherwise (default value)} \end{cases}$

2.2 $n+1$: $\langle y_1, \ldots, y_n \rangle \leftarrow F(x_1', \ldots, x_n')$

2.3 $(1 \le i \le n)$ $n+1 \to i$: y_i

2.4 $(1 \le i \le n)$ i: **output** y_i

Fig. 2. Ideal protocol for trusted host $(n+1)$ to compute probabilistic multivalued function F, which without loss of generality is extended to handle default values. When default values are used, the trusted host returns an n-bit vector with a 1 in the ith position if player i supplied a value outside the domain (not shown).

fact, the problem of designing secure multiparty protocols is motivated by the lack of, and great risk inherent in, trusted parties. A simple, ideal protocol represents the goal to pursue: compute what a trusted host would compute when supplied with inputs from each participant, even though such a trusted host is never available. Figure 2 describes a two-round *ideal* protocol to compute a function $F \in$ PFF.

2.4. *Concatenating General Protocols*

Auxiliary inputs are included for two reasons. The first is of lesser concern in this paper. In computationally bounded models, auxiliary inputs present a means to introduce nonuniformity to uniform Turing machine computations. Often, cryptographic techniques are claimed to be secure against nonuniform circuit families used to break them. Auxiliary inputs allow strong measures of security to be considered when limited resources are of concern.

The second reason is the more important: auxiliary inputs provide information gained externally, often through earlier interactions. Auxiliary inputs for nonfaulty players capture historical information from earlier protocols. When an adversary corrupts a player in a later protocol, it should gain *all* the historical information held by that player. Furthermore, a protocol that is secure for all possible auxiliary inputs need not be concerned with being broken by sensitive information revealed in advance. In other words, when one protocol follows another, the resilience of the second protocol is not jeopardized by information from the earlier one.

Let $\{\alpha_1, \alpha_2, \ldots\}$ be a collection of protocols, and let $f(n, m, k)$: $\mathbf{N} \to \mathbf{N}$. The *sequential concatenation* of $f(n, m, k)$ protocols from $\{\alpha_r\}$ is the protocol described in Fig. 3 and is written $\alpha^f = \circ\alpha_r$. The function $\mathbf{ExecA}_r$ represents an execution of

CONCAT-PROTOCOL($\{\alpha_r\}, f(n, m, k)$)

1 $(1 \le i \le n)$ i: $x_i(1) \leftarrow x_i$; $a_i(1) \leftarrow a_i$

2 **For** $r = 1..f(n, m, k)$ **do**

Run protocol α_r to obtain new global state:

$\langle x_1(r+1), \ldots, x_n(r+1) \rangle \cdot \langle a_1(r+1), \ldots, a_n(r+1) \rangle \cdot a_A(r+1) \leftarrow$

$\mathbf{ExecA}_r(n \# m \# \langle x_1(r), \ldots, x_n(r) \rangle \cdot \langle a_1(r), \ldots, a_n(r) \rangle \cdot a_A(r), k)$

3 $(1 \le i \le n)$ i: **output** $x_i(f(n, m, k)+1)$

Fig. 3. Algorithmic specification of the concatenation of $f(n, m, k)$ protocols from the collection $\{\alpha_1, \alpha_2, \ldots\}$.

protocol α_r. The inputs, output, and view of player i in the rth subprotocol are denoted $x_i(r)$, $a_i(r)$, $y_i(r)$, and $v_i(r)$. At each execution, $x_i(r+1)$ is set to the previous output, $y_i(r)$. The view of the current subprotocol contains the auxiliary input from the previous protocol, so that $a_i(r+1)$ becomes the concatenation $y_i(r)$ of $a_i(r)$ with the messages seen and computations performed by i during the subprotocol.

The adversary is the same throughout the entire execution, and the set T of players it corrupts must contain those it corrupts in each subprotocol. More formally, if subprotocol α_r is associated with adversary class A_r, then the adversary class for $\circ\alpha_r$ is $\bigcap \mathscr{A}_r$. The adversary may be regarded as a repeated operation of $\mathscr{A}$ with auxiliary input $a_A(r+1)$ updated to $y_A(r)$ each time.

3. Security

In the past, the analysis of the security and fault-tolerance of multiparty tprotocols has pursued the following sort of reasoning: a trusted host would return the value of $f(x_1, \ldots, x_n)$ but no other information, so a general protocol should maintain *privacy*, i.e., it should reveal no more than $f(x_1, \ldots, x_n)$; a trusted host would return the right value, maintaining *correctness*; a trusted host would collect all inputs individually before computing $f(x_1, \ldots, x_n)$ or returning *any* information at all, maintaining *independence of inputs*, i.e., requiring that one processor's choice of input[2] is not influenced by another's. Addressing these properties separately and in an *ad hoc* manner introduces confusion and makes formal definitions difficult. For example, correctness must be defined with respect to inputs, and often a notion of commitment to inputs is introduced to facilitate this approach. A commitment scheme (encryption and broadcast of inputs, for example) may interfere with independence of inputs (a faulty processor may choose the *encryption* of a good processor's input as its own input, with unpredictable effects on independence—or at least effects that are complicated to analyze). Without becoming mired in historical details, suffice it to say that the *ad hoc*, analytic approach allows a variety of different versions of definitions that are not overly cohesive, comprehensible, or comparable.

To unify these otherwise diverse definitions, we must return to the simplicity of the original goal, that of achieving the same results as though an *ideal*, trusted party were available. Galil *et al.* [19] and later Beaver and Rogaway *et al.* [6], [30], [12] make an initial foray in this direction, measuring the *information* leaked by a protocol with reference to a *fault oracle* that is guaranteed to return exactly one computation $f(x_1, \ldots, x_n)$ based on inputs held by correct processors and arbitrarily modified inputs of faulty processors.[3] In analogy to the zero-knowledge approach pioneered by Goldwasser *et al.* [23], we must demonstrate that the transcript of an

[2] A *choice* of inputs is necessary, e.g., when flipping a global coin by computing the parity of random input bits supplied by each processor, or in tallying a secret ballot.

[3] Even in an ideal case, faulty processors can supply different inputs to the trusted host; but this is their only influence.

execution of the general protocol can be generated ("simulated") using only the information provided by restricted access to the fault oracle.

Kilian *et al.* [26] and Beaver [6], [5] note that it is not enough to measure the *information* obtained by the adversary. The *influence* the adversary has on the final output through calculated choice of corrupted players and their inputs, must also be considered. Correctness and dependencies among inputs are closely related to the power of influence. They propose that a simulation of a general protocol should not only produce a transcript seen by an adversary but should induce *outputs* for the correct players that are identical (indistinguishable) with those in the general protocol. Because the fault-oracle computes all outputs correctly, this lends the additional notion of *correctness* to the oracle-based simulation approach, and unifies widely varying definitions.

Though the fault-oracle approach can be modified to capture the combination of information and influence, it is rather inelegant. It does not support modular proofs. For example, two protocols may each be proved secure by relating each to a different fault-oracle. To prove the concatenation of the two protocols secure, we must use a single fault-oracle. It is not clear how two calls to two oracles can be captured by a single call to a single oracle. If the concatenation is to be proved secure without further modification to the definitions, we must begin from scratch, at best able to employ the particular details of the particular individual proofs in an *ad hoc* manner.

3.1. *Relative Resilience: The Tool To Measure Security*

The insight that surpasses the solid but unsatisfactory fault-oracle approach is that a means to compare *any* two protocols, not simply a given protocol against a fault-oracle, is essential to providing comprehensible definitions and proofs. In his doctoral dissertation, the author [6] introduces and defines the notion of *relative resilience*. The work presented here adopts that approach. Given a means to compare the resilience of any two protocols, *absolute resilience* can be defined with respect to a standard, particular, *ideal* protocol, in which an incorruptible, trusted host is available. A general protocol is resilient if and only if it is as resilient as the ideal protocol.

Consider two protocols, α and β, each with an associated class of allowable adversaries, A_α and A_β. To compare the resilience of protocol α against an adversary $\mathscr{A} \in \mathsf{A}_\alpha$ to the resilience of protocol β, adversary $\mathscr{A}$ should be allowed to wreak havoc on protocol β. Unfortunately, α and β may be radically different protocols. One might have many more players than the other, one might disallow certain players from being corrupted, one might be written in *C* while the other is written in FORTRAN, and so forth. We cannot simply run protocol β with adversary $\mathscr{A}$.

Instead, $\mathscr{A}$ is surrounded by an *interface*, $\mathscr{I}$, that creates an environment for $\mathscr{A}$ similar to that it finds in protocol α, and that at the same time participates in protocol β as an adversary of its own right. The pair of machines $\mathscr{I}$ and $\mathscr{A}$ may be considered as a single adversary $\mathscr{I}(\mathscr{A})$ when the communications between the two are ignored. Of course, the combination $\mathscr{I}(\mathscr{A})$ must be a permissible adversary in

A_β.[4] In a certain sense, the interface translates all of the adversary's prowess in corrupting protocol α into a corruption of β.

Definition 10. An *interface* $\mathscr{I}$ is a machine (interactive Turing machine or general player) with two communication lines; the first is called an *environment simulation* line, and the second is called an *adversarial* line. If α admits adversary class A_α and β admits adversary class A_β, then $\mathscr{I}$ is an *interface from* α *to* β if for every $\mathscr{A} \in \mathsf{A}_\alpha$, $\mathscr{I}(\mathscr{A}) \in \mathsf{A}_\beta$.

To say that α is as resilient as β is to say that α withstands attacks as well as β. Thus, there must be some interface $\mathscr{I}$ that translates attacks on α into attacks on β, such that the information gained and influence wielded by adversary $\mathscr{A}$ in α is the same as that of $\mathscr{I}(\mathscr{A})$ in β.

Definition 11 (Relative Resilience). A protocol α is *as resilient as* protocol β with respect to adversary classes A_α and A_β, written

$$\alpha \succeq_{(\mathsf{A}_\alpha, \mathsf{A}_\beta)} \beta,$$

if there exists an interface $\mathscr{I}$ from α to β such that for all adversaries $\mathscr{A} \in \mathsf{A}_\alpha$,

$$[\alpha, \mathscr{A}] \approx [\beta, \mathscr{I}(\mathscr{A})]$$

The subscript $(\mathsf{A}_\alpha, \mathsf{A}_\beta)$ is omitted where clear from context. If $[\alpha, \mathscr{A}] \approx^{\delta(k)} [\beta, \mathscr{I}(\mathscr{A})]$, the protocols are called $\delta(k)$-relatively resilient, written $\alpha \succeq^{\delta(k)} \beta$. The protocols are *exponentially* ($\succeq^e$) or *statistically* ($\succeq^s$) relatively resilient according to how indistinguishable the ensembles are.

It is not only *information* but *influence* on outputs that is important. Previous attempts to define security seem quite *ad hoc* because they treat these issues—privacy and correctness—separately. The definitions presented here unify all properties *a priori*. Furthermore, these definitions provide a *general* approach, allowing comparisons between arbitrary protocols, not simply direct measurements against a fixed standard. This permits modular proof techniques: the property of relative resilience (and its component properties, privacy and correctness) is easily shown to be a partial order. (It is not clear that it is an equivalence class: the existence of an interface in one direction is not clearly sufficient to show the existence of an interface for the reverse direction.) Hence proofs can be simplified by comparing intermediate protocols rather than proceeding in one intricate step to the goal.

A standard of secure computation is nonetheless necessary. The ideal protocol serves as the measure of *absolute* resilience, in the same way that a trusted party guaranteed to provide a correct statement, "$x \in L$," provides the standard for zero-knowledge proofs.

The diagram in Fig. 4 illustrates a comparison of resilience between a real protocol and an ideal protocol. Three scenarios are of importance. The first represents the ideal world, in which a trusted and reliable host is available and

[4] Technically, there must be an adversary in A_β with an identical input/output behavior as that produced by the pair of players, $\mathscr{I}$ and $\mathscr{A}$.

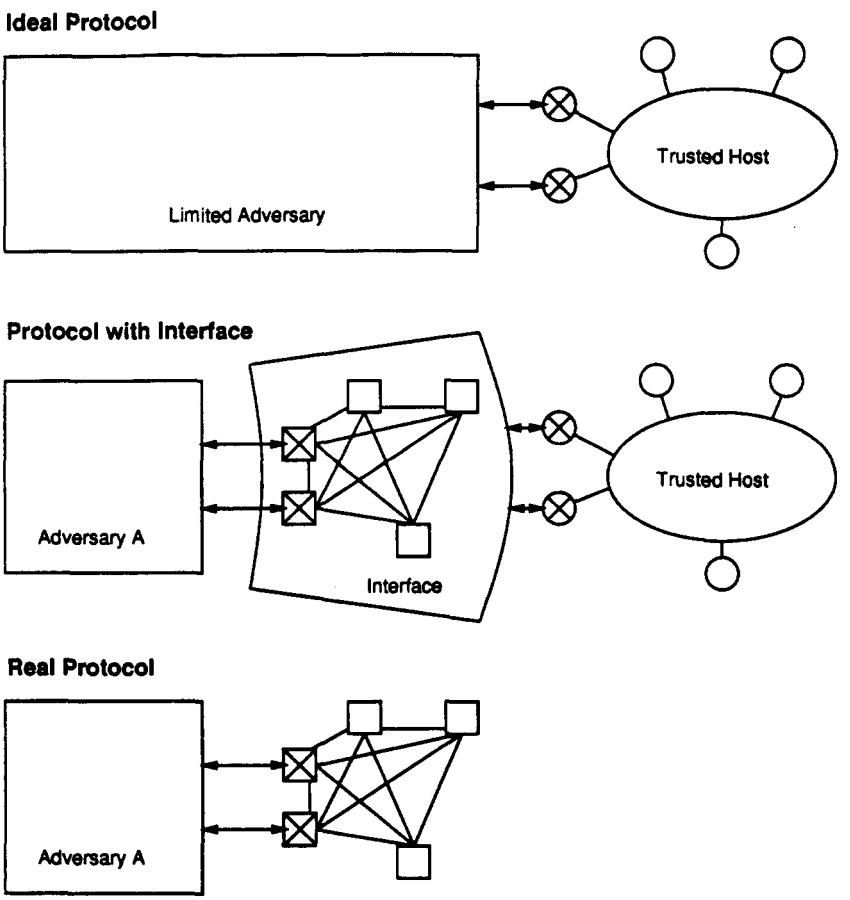

Fig. 4. Three scenarios: first, an ideal protocol with a trusted host, clearly delineating the limited information and influence of an adversary; second, an adversary attacking a trusted host by way of an interface that creates a simulated environment for it; third, an adversary attacking a real protocol with no trusted parties. Squares and circles indicate players; squares within the interface are simulated players. The cross (×) marks indicate corrupted players.

ensures that the adversary is truly restricted to gaining only the inputs and outputs of players of its choice. The third represents the real world, in which no player can be trusted but a protocol must be designed to perform the same computations as in the ideal world, correctly and privately. The second and intermediate scenario joins the two, modeling the interaction in an ideal protocol attacked by a real adversary that is assisted by an interface. The information and influence of the adversary, allowed to attack the ideal protocol as best it may, is clearly delineated in this central case. It connects the clear measurements of security and reliability in the ideal case to the less easily understood powers of the adversary in the real world.

Definition 12 (Resilience). A *t-resilient protocol for F* is a protocol Π satisfying:

$$\Pi \succeq_{(A^t, A^f_{ideal})} \mathsf{ID}(F).$$

A protocol is *exponentially* or *statistically resilient* according to whether it is exponentially or statistically as resilient as $\mathsf{ID}(F)$. Another sort of resilience, *computational resilience*, in which the appropriate ensembles are computationally indistinguishable, comes into play in resource-bounded settings, which are not considered here. This work focuses on achieving exponential resilience.

Remark. Though this paper treats resilience as a whole, the property of resilience can be broken down into components, the most important of which are *privacy* and *correctness*. Historically, the division into separate properties has caused many difficulties in understanding and proving security, not the least of which has been to find a compatible set of definitions. Resilience, however, provides a unified and simple pair of definitions:

Definition 13. A protocol α is *as private as* protocol β if there exists an interface $\mathscr{I}$ from α to β such that, for all adversaries $\mathscr{A} \in \mathsf{A}_\alpha$,

$$[\alpha, \mathscr{A}]^{Y_A} \approx [\beta, \mathscr{I}(\mathscr{A})]^{Y_A}.$$

A protocol α is *as correct as* protocol β if there exists an interface $\mathscr{I}$ from α to β such that, for all adversaries $\mathscr{A} \in \mathsf{A}_\alpha$,

$$[\alpha, \mathscr{A}]^{Y_{[n]}} \approx [\beta, \mathscr{I}(\mathscr{A})]^{Y_{[n]}}.$$

A protocol is *t-private* if it is as private as the ideal protocol, and it is *t-correct* if it is as correct as the ideal protocol. Note that the individual properties of privacy and correctness do not imply resilience unless the interfaces used to demonstrate both are the same.

3.2. *Fault Recovery and Defective Runs*

Cheating is defined not as departure from the protocol but as producing messages that are not *consistent* with some possible computation by a valid player. In other words, the adversary may replace messages from corrupted players, but if the set of messages has nonzero probability of being produced by a nonfaulty player, then the adversary has not cheated. There is no way to tell whether a player has generated a string according to the proper distribution or not, apart from checking whether that string is *possible* or not. Thus, a faulty player asked to supply a uniformly random bit may instead supply a biased bit without a problem, but if it supplies the string 00110, for example, it is deemed to cheat.

Definition 14. An adversary *cheats* on message $\mu(i, j, r)$ from faulty player i if a nonfaulty player i would never send $\mu(i, j, r)$; that is, if $\Pr[\mu(i, j, r) | \mu(i, [n], 1..r-1), \mu([n], i, 1..r-1)] = 0$.

Most protocols turn out to be robust against using improper *distributions*, though this property is not always explicitly mentioned. For example, a protocol to generate a uniformly random bit will request bits from all players; if at least one player

supplies a uniformly random bit, the parity will be uniformly random, despite substituted distributions, even if no player accuses another of cheating.

The difficult part in designing protocols is to detect cheating and recover gracefully. A relatively simple recovery method uses secret sharing: each player must share its inputs and random choices at the start of the protocol. During the protocol, a protocol RECOVER is run whenever cheating is detected to reconstruct the secrets of cheating players; thereafter, the messages from the cheating players are implicitly determined, publicly known and locally computable, and need not actually be sent. A slightly more complicated method is to run a recovery protocol that converts the n-player protocol to an $(n-1)$-player protocol, redistributing the information held by the cheating player without revealing it. The same techniques enabling secret addition allow such a recovery protocol. As in a protocol by Galil *et al.* [18], this preserves privacy of participants suffering from accidental faults. In the case of polynomial-based secret sharing, this consists of reducing the degree of each polynomial by 1. In the protocols of this paper, it is implicitly assumed that RECOVER is executed when cheating is detected; to state this explicitly in all protocol descriptions would be burdensome.

A *defective run* of a protocol is defined to be an execution in which the adversary cheats using some player i that is *not* immediately deemed by other players to be cheating. Normally, an interface easily determines whether a run is defective, by calculating whether the messages from $\mathcal{A}$ indicate cheating and whether the properly distributed responses from nonfaulty players (which the interface simulates) include messages disqualifying the cheating player. Defective runs are ordinarily a problem for the interface because they may permit $\mathcal{A}$ greater information and influence in protocol α than it ought to have, whereas the interface has neither the information nor the degree of influence on players in its own protocol (β) that $\mathcal{A}$ has in α. It will be shown that if the probability of a defective run is small, then the behavior of the interface when faced with a defective run is irrelevant.

Ideal protocols are implicitly assumed to return an extra result, namely a string of n bits set to 1 for each player that sends the trusted host an improper message. Thus, corrupted players may choose not to participate (i.e., to cheat), but this choice may become public information.

3.3. *Threshold Schemes*

Given formal definitions for security, threshold schemes (see secret sharing as sketched in Section 1) can now be defined. Threshold schemes are concerned with creating distributed representations of values that cannot be changed and that leak no information.

A *robust* representation of a value can be decoded regardless of attempts to change portions of the representation. Error-correcting codes and verifiable secret sharing are examples. If $\vec{y}$ is a vector of n values, a *t-modification* of $\vec{y}$ is a vector differing in at most t places from $\vec{y}$. Let S be some set of values.

Definition 15. A (probabilistic) function **sha**: $S \to \text{dist}(2^{(\Sigma^*)^n})$ is a *t-robust representation* if there exists a function **rec** such that $\mathbf{rec}(\vec{y}') = s$, for all $s \in S$ and for all t-modifications $\vec{y}'$ of any $\vec{y}$ for which $\Pr_{\mathbf{sha}(s)}[\vec{y}] > 0$.

The *ideal vacuous protocol*, denoted $\mathsf{ID}(0)$, returns no outputs apart from a vector listing players who have decided to become disqualified. Clearly, a vacuous protocol satisfies all intuitions regarding privacy, so any protocol that is as private must be absolutely private.

Definition 16. A function **sha** is *t-private* if the ideal protocol $\mathsf{ID}(\mathbf{sha})$ to compute it is as private as the ideal vacuous protocol, $\mathsf{ID}(0)$.

Robustness and privacy together define secret sharing:

Definition 17. A *threshold scheme* with threshold t is a pair of protocols SHA and REC, where SHA computes a t-robust and t-private representation **sha**, and REC computes the corresponding function **rec**.

4. Techniques for Proving Security

Several lemmas provide a foundation for *proving* the security of the protocols presented in this paper. These range from simple lemmas regarding composition of probabilistic functions to previously unproven folk theorems whose informal statements, without care for technical detail, are often false. No algorithms are presented in this section.

4.1. *Ensembles*

Replacing an ensemble in a composition of several ensembles by one that is $\delta(k)$-indistinguishable gives rise to a composite ensemble that is at most $\delta(k)$-indistinguishable.

Lemma 1. *Let P_1, P_2^α, P_2^β, and P_3 be ensembles. Define the ensembles*

$$Q^\alpha(z, k) = \{z_1 \leftarrow P_1(z, k);\ z_2^\alpha \leftarrow P_2^\alpha(z_1, k);\ z_3^\alpha \leftarrow P_3(z_2^\alpha, k):\ z_3^\alpha\},$$

$$Q^\beta(z, k) = \{z_1 \leftarrow P_1(z, k);\ z_2^\beta \leftarrow P_2^\beta(z_1, k);\ z_3^\beta \leftarrow P_3(z_2^\beta, k):\ z_3^\beta\}.$$

If $P_2^\alpha \approx^{O(\delta(k))} P_2^\beta$, then $Q^\alpha \approx^{O(\delta(k))} Q^\beta$. The convergence parameters are identical.

Proof. Let k_0 be the convergence parameter for P_2^α and P_2^β, with associated constant c_0. Assume by way of contradiction that $Q^\alpha \not\approx^{O(\delta(k))} Q^\beta$. Then there is a $k \geq k_0$ and a z such that

$$\sum_{z_1, z_2, z_3} |\mathrm{Pr}_{P_3(z_2,k)}[z_3]\mathrm{Pr}_{P_2^\alpha(z_1,k)}[z_2]\mathrm{Pr}_{P_1(z,k)}[z_1] - \mathrm{Pr}_{P_3(z_2,k)}[z_3]\mathrm{Pr}_{P_2^\beta(z_1,k)}[z_2]\mathrm{Pr}_{P_1(z,k)}[z_1]| > c_0 \cdot \delta(k).$$

Thus there exists a z_1 such that

$$\sum_{z_2, z_3} |(\mathrm{Pr}_{P_3(z_2,k)}[z_3]\mathrm{Pr}_{P_2^\alpha(z_1,k)}[z_2] - \mathrm{Pr}_{P_3(z_2,k)}[z_3]\mathrm{Pr}_{P_2^\beta(z_1,k)}[z_2])| > c_0 \cdot \delta(k).$$

It follows that

$$\sum_{z_2} |\Pr_{P_2^\alpha(z_1,k)}[z_2] - \Pr_{P_2^\beta(z_1,k)}[z_2]|$$

$$= \sum_{z_2} \left(\sum_{z_3} \Pr_{P_3(z_2,k)}[z_3] \cdot |\Pr_{P_2^\alpha(z_1,k)}[z_2] - \Pr_{P_2^\beta(z_1,k)}[z_2]| \right) > c_0 \cdot \delta(k).$$

Since $k \geq k_0$, this contradicts $P_2^\alpha \approx^{O(\delta(k))} P_2^\beta$. □

4.2. *Relative Resilience Is Transitive*

Theorem 2. *Relative resilience is reflexive and transitive. This holds for perfect, exponential, and statistical resilience.*

Proof. Reflexivity is trivial, using an interface that simply passes corruption requests and responses back and forth without changing them. For transitivity (including the cases of perfect, exponential, and statistical relative resilience), it suffices to show that $\alpha \succeq^{\delta_1(k)} \beta$ and $\beta \succeq^{\delta_2(k)} \gamma$ implies $\alpha \succeq^{\delta_1(k)+\delta_2(k)} \gamma$. If $\alpha \succeq^{\delta_1(k)} \beta$ and $\beta \succeq^{\delta_2(k)} \gamma$, there exist interface $\mathscr{I}_{\alpha,\beta}$ and $\mathscr{I}_{\beta,\gamma}$ such that for all $\mathscr{A}$, $[\alpha, \mathscr{A}] \approx^{\delta_1(k)} [\beta, \mathscr{I}_{\alpha,\beta}(\mathscr{A})]$ and for all $\mathscr{A}'$, $[\beta, \mathscr{A}'] \approx^{\delta_2(k)} [\gamma, \mathscr{I}_{\beta,\gamma}(\mathscr{A}')]$. Define $\mathscr{I}_{\alpha,\gamma} = \mathscr{I}_{\beta,\gamma} \circ \mathscr{I}_{\alpha,\beta}$; this interface passes requests from $\mathscr{A}$ to an internal copy of $\mathscr{I}_{\alpha,\beta}$, which then requests corruptions from an internal copy of $\mathscr{I}_{\beta,\gamma}$, whose requests are then output by $\mathscr{I}$. The responses are passed through the internal machines in the reverse direction. Now,

$$[\alpha, \mathscr{A}] \approx^{\delta_1(k)} [\beta, \mathscr{I}_{\alpha,\beta}(\mathscr{A})] \approx^{\delta_2(k)} [\gamma, \mathscr{I}_{\beta,\gamma}(\mathscr{I}_{\alpha,\beta}(\mathscr{A}))] = [\gamma, \mathscr{I}_{\alpha,\gamma}(\mathscr{A})].$$

It is straightforward to show that $[\alpha, \mathscr{A}] \approx^{\delta_1(k)+\delta_2(k)} [\gamma, \mathscr{I}_{\alpha,\gamma}(\mathscr{A})]$, hence $\alpha \succeq^{\delta_1(k)+\delta_2(k)} \gamma$. □

4.3. *Postprotocol Corruption*

For proofs involving dynamic adversaries, it is essential that the interface be able to generate the view of a newly corrupted player. Certainly, during the run of a protocol, an interface must be able to compute such a view accurately, in order to produce a reasonable facsimile of the **Fault** function for $\mathscr{A}$. In the case of the sequential concatenation of protocols, however, the view of a reliable player includes its view during previous protocols. An interface $\mathscr{I}$ that runs subinterfaces for each subprotocol does not, during the rth subprotocol execution, obtain $x_i(r)$ and $a_i(r)$ when it requests the corruption of i; it receives only $x_i(1)$ and $a_i(1)$. Subinterface $\mathscr{I}_r$ requires an answer containing $x_i(r)$ and $a_i(r)$, however. The view from earlier protocols determines both values, but $\mathscr{I}$ must generate this view.

If $\mathscr{I}$ could request of $\mathscr{I}_{r-1}$ that player i be corrupted, then it would obtain a view specifying $x_i(r)$ (the output of protocol α_{r-1}) and $a_i(r)$ (containing $a_i(r-1)$ and the view of player i in protocol α_{r-1}). Unfortunately, even though $\mathscr{I}_{r-1}$ may be ready and willing to accept more requests on its adversarial line, as though protocol α_{r-1} were in its final stage when the adversary is still able to corrupt players after all

communications have been sent, there is no guarantee that its responses are accurate. The fact that it is a good interface for the previous protocol means that it is guaranteed only to generate good responses for requests made by adversaries to that protocol, not necessarily for arbitrary requests made later by other, perhaps more powerful and extensive, adversaries. For example, in a computationally bounded setting, there is no reason an interface that works for coalitions generated by polynomial-time adversaries will also work for *arbitrary* coalitions, including those chosen later.

Fortunately, in the cases of perfect or exponential resilience, static adversaries, or memoryless protocols (in which views can be erased), the interface must of necessity satisfy this requirement. To see this in the case of perfect or exponential resilience, note that for any $\mathscr{A}$ there is an $\mathscr{A}'$ that, in the final round, selects an additional player randomly and requests its corruption. The probability of selecting a given player is at least $1/n$, so if the interface fails to provide a $\delta(k)$-accurate view, the resulting ensemble will be at least $(\delta(k)/n)$-distinguishable from the desired ensemble.

To define postprotocol corruption, we allow an adversary to attack Π even after its execution is complete. A vacuous protocol $\Pi(\varnothing)$ that has *no* channels and executes *no* instructions serves as a convenient tool to describe an attack on a network after all computations have finished. Consider the following special execution of Π and $\Pi(\varnothing)$ with adversaries $\mathscr{A}$ and $\hat{\mathscr{A}}$: adversary $\mathscr{A}$ attacks Π, adversary $\hat{\mathscr{A}}$ is given $\vec{y} \cdot y_A$, and finally $\hat{\mathscr{A}}$ attacks $\Pi(\varnothing)$. In the attack on $\Pi(\varnothing)$, the "postprotocol adversary" $\hat{\mathscr{A}}$ has access to the adversary output y_A and state q_A, the original auxiliary input a_A, the player outputs $\vec{y}$, and some subset of the views $\vec{v}$ generated by the first execution $\mathbf{ExecA}[\Pi, \mathscr{A}]$. It then requests more corruptions, receiving outputs and views of protocol Π. Define $[\Pi, \mathscr{A}, \hat{\mathscr{A}}]$ to be $\vec{y} \cdot y_A \cdot y_{\hat{A}}$, where $y_{\hat{A}}$ is the output of $\hat{\mathscr{A}}$.

A postprotocol corrupter is like an interface: it translates views, inputs, and outputs of players corrupted in one protocol (β) to facsimiles of views, inputs, and outputs of players in another (α).

Definition 18. A *postprotocol corrupter* $\hat{\mathscr{I}}$ is an interface that, when given corruption request i, makes a request for the input and view of player i on its adversarial line and returns a constructed input and view on its environment simulation line. A corrupter *from* α *to* β satisfies $(\forall \hat{\mathscr{A}} \in \mathsf{A}_\alpha)\ \hat{\mathscr{I}}(\hat{\mathscr{A}}) \in \mathsf{A}_\beta$.

It is often convenient to imagine that both $\mathscr{I}$ and $\mathscr{A}$ continue running after β is finished, with $\mathscr{I}$ executing $\hat{\mathscr{I}}$ and $\mathscr{A}$ executing $\hat{\mathscr{A}}$ (after being given the string $\vec{y}$); then we may "make corruption requests of $\mathscr{I}$ after β is finished."

Define the ensemble $[\beta, \mathscr{I}(\mathscr{A}), \hat{\mathscr{I}}(\hat{\mathscr{A}})]$ as the outputs $\vec{y} \cdot y_A \cdot y_{\hat{A}}$ induced by the following execution: $\mathscr{A}$ attacks β through $\mathscr{I}$, $\hat{\mathscr{A}}$ is given $\vec{y} \cdot \vec{y}_A$, and finally $\hat{\mathscr{A}}$ attacks $\Pi(\varnothing)$ through $\hat{\mathscr{I}}$, who is given the final state of $\mathscr{I}$ as an input. We would like to say that, even knowing all the outputs of the protocol, a postprotocol adversary cannot tell whether it is receiving outputs and views from α or outputs and views generated by a postprotocol corrupter $\hat{\mathscr{I}}$:

Definition 19. Protocol α is *postprotocol corruptible* with respect to protocol β if there exists an interface $\mathscr{I}$ from α to β and a postprotocol corrupter $\hat{\mathscr{I}}$ from α to β such that, for all $\hat{\mathscr{A}}$, $\mathscr{A} \in \mathsf{A}_\alpha$,

$$[\alpha, \mathscr{A}, \hat{\mathscr{A}}] \approx [\beta, \mathscr{I}(\mathscr{A}), \hat{\mathscr{I}}(\hat{\mathscr{A}})].$$

Remark. $\hat{\mathscr{I}}$ is *not* allowed to see all the outputs $\vec{y}$ from α; but it may see some of them, since it can request new corruptions from β. In the case of t-adversaries, the postprotocol adversary $\hat{\mathscr{A}}$ is still limited to requesting corruptions of t-subsets (despite having *all* the outputs $\vec{y}$—but not the views $\vec{v}$); $\hat{\mathscr{I}}$ can obtain the necessary outputs through corruption, and its real job is to transform *views* v_i^β of β into views v_i^α apparently of α.

4.4. *Composition and Concatenation*

Under certain circumstances, the concatenation of polynomially many resilient protocols is itself resilient. Consider first a family $\{P_i\}$ of ensembles. With respect to function $f: \mathbf{N} \to \mathbf{N}$, define the *composition* P^f *of* $f(k)$ *ensembles from* $\{P_i\}$ by $P^f(z, k) = P_{f(k)}(P_{f(k)-1}(\cdots P_1(z, k)\cdots), k)$. Two ensemble families $\{P_i\}$ and $\{Q_i\}$ are *uniformly pairwise* $\{\delta_i\}$*-indistinguishable*, written $\{P_i\} \asymp^{\delta_i} \{Q_i\}$, if there exists a k_0 such that, for all i, $P_i \approx^{\delta_i(k)} Q_i$ with convergence parameter at most k_0. Another way to state this is

$$(\exists k_0) \quad (\forall i) \quad (\forall k \geq k_0) \quad (\forall z \in \Sigma^*) \qquad |P_i(z, k) - Q_i(z, k)| < \delta_i(k). \tag{1}$$

The order of quantifiers $(\exists k_0)$, $(\forall i)$ is essential.

Lemma 3. *The composition of indistinguishable ensembles is indistinguishable*: $\{P_i\} \asymp^{\delta_i} \{Q_i\} \Rightarrow P^f \approx^{\delta} Q^f$, *where* $\delta(k) = \sum_{i=1}^{f(k)} \delta_i(k)$.

Proof. Assume $\{P_i\} \asymp^{\delta_i} \{Q_i\}$ but $P^f \not\approx^{\delta} Q^f$. Let k_0 be the uniform convergence parameter (see (1)) for the former. For some $k \geq k_0$ and for some z, defining $R_l = P_{f(k)} \circ \cdots \circ P_l \circ Q_{l-1} \circ \cdots \circ Q_1$,

$$\begin{aligned}
\sum_{i=1}^{f(k)} \delta_i(k) &< \sum_z |P_{f(k)} \circ \cdots \circ P_1(z, k) - Q_{f(k)} \circ \cdots \circ Q_1(z, k)| \\
&\leq \sum_z |R_0 - R_1 + R_1 - R_2 + \cdots + R_{f(k)-1} - R_{f(k)}| \\
&\leq \sum_z \sum_{i=1}^{f(k)} |R_{i-1}(z, k) - R_i(z, k)|.
\end{aligned}$$

Reversing the order of summation, it follows that, for some i, $\delta_i(k) < \sum_z |R_{i-1}(z, k) - R_i(z, k)|$. By Lemma 1, $\delta_i(k) < \sum_z |P_i(z, k) - Q_i(z, k)|$, contradicting $\{P_i\} \asymp^{\delta_i} \{Q_i\}$. □

Now consider a family $\{\alpha_i\}$ of protocols, and define the *concatenation* α^f *of* $f(k)$ *protocols from* $\{\alpha_i\}$ by $\alpha^f(n, m, k)(\vec{x} \cdot \vec{a} \cdot a_A) = \alpha_{f(k)} \circ \cdots \circ \alpha_1(n, m, k)(\vec{x} \cdot \vec{a} \cdot a_A)$. Two

protocol families $\{\alpha_i\}$ and $\{\beta_i\}$ are *uniformly pairwise relatively* $\{\delta_i\}$*-resilient*, written $\{\alpha_i\} \trianglerighteq \{\beta_i\}$, if there exists a k_0 such that, for all i, $\alpha_i \succeq^{\delta_i(k)} \beta_i$ with convergence parameter at most k_0, and each α_i is postprotocol corruptible with respect to β_i.

Lemma 4. *If* $\{\alpha_i\}$ *and* $\{\beta_i\}$ *are uniformly pairwise relatively resilient and postprotocol corruptible, then the concatenations* α^f *and* β^f *are also relatively resilient and postprotocol corruptible:* $\{\alpha_i\} \trianglerighteq^{\delta_i} \{\beta_i\} \Rightarrow \alpha^f \succeq^{\delta} \beta^f$, *where* $\delta(k) = \sum_{i=1}^{f(k)} \delta_i(k)$.

Proof. Let $\mathscr{A}$ be an arbitrary adversary in A_α. For each $i \geq 1$, let interface $\mathscr{I}_i$ satisfy $(\forall \mathscr{A} \in \mathsf{A}_{\alpha_i})\ [\alpha_i, \mathscr{A}] \approx^{\delta_i} [\beta_i, \mathscr{I}(\mathscr{A})]$, and let postprotocol corrupter $\hat{\mathscr{I}}_i$ satisfy $(\forall \hat{\mathscr{A}}, \mathscr{A} \in \mathsf{A}_{\alpha_i})\ [\alpha_i, \mathscr{A}, \hat{\mathscr{A}}] \approx^{\delta_i} [\alpha_i, \mathscr{I}_i(\mathscr{A}), \hat{\mathscr{I}}_i(\hat{\mathscr{A}})]$. Let these ensembles converge pairwise uniformly with convergence parameter k_0. Define $\mathscr{I}$ to be an interface from α^f to β^f that, on input k, runs interfaces $\mathscr{I}_1, \ldots, \mathscr{I}_{f(k)}$, running the attack of $\mathscr{A}$ on α_i synchronously with the attack of $\mathscr{I}_i$ on β_i. That is, $\mathscr{I}$ expects a series of corruption requests from adversary $\mathscr{A}$ corresponding to executions of α_1 up to α_f; while $\mathscr{A}$ is generating requests for corruptions in α_i, $\mathscr{I}$ participates in β_i, running $\mathscr{I}_i$ as a subroutine.

For the first subprotocol, $\mathscr{I}$ runs $\mathscr{I}_1$, using its corruption requests to corrupt players in β and supplying $\mathscr{I}_1$ with the information it receives (i.e., $(x_j^\beta(1), a_j^\beta(1))$). Now, note that even though an interface induces player outputs $\vec{x}^\beta(2)$ and an adversary output $a_A^\beta(2)$ that are indistinguishable from those seen in an α_1-run, the views $\vec{a}^\beta(2)$ of nonfaulty players after an execution of β_1 will be different from those of nonfaulty players after an execution of α_1. Thus, the fact that $\mathscr{I}_2$ is a good interface will not help during the second subprotocol, since α_2 and β_2 must be started with the same inputs and auxiliary inputs in order for $\mathscr{I}_2$ to apply. Fortunately, since nonfaulty players compute independently of their auxiliary inputs (see Definition 5), we can run β_2 using incorrect auxiliary inputs for nonfaulty players, as long as when these auxiliary inputs are revealed by corruption, the interface replaces them with a good facsimile of an α_1-view.

Thus, when $i > 1$, $\mathscr{I}$ filters the responses to corruption requests made by $\mathscr{I}_i$, so that $\mathscr{I}_i$ see auxiliary inputs containing views apparently from previous α-protocols $\alpha_1, \ldots, \alpha_{i-1}$, instead of the ones from $\beta_1, \ldots, \beta_{i-1}$ that are returned to $\mathscr{I}$ from its corruption of β. When $\mathscr{I}_i$ requests the corruption of player j, $\mathscr{I}$ corrupts j in β. If player j is already corrupt, then $\mathscr{I}$ has already generated its input and auxiliary input, and it need only tack on the view of the current run of β_i before it replies to $\mathscr{I}_i$. Otherwise, $\mathscr{I}$ generates a view of previous α-subprotocols by using postprotocol corruption. Interface $\mathscr{I}$ obtains $a_j^{\beta_1}(1), \ldots, a_j^{\beta_{i-1}}(i-1)$ from the corruption of j in β_i; since α_1 and β_1 were started on the same inputs and auxiliary inputs, $\mathscr{I}$ sets $a_j(1) = a_j^\beta(1)$. To obtain $a_j(2)$, $\mathscr{I}$ requests postprotocol corruption of player j from $\hat{\mathscr{I}}_1$, and it supplies $\hat{\mathscr{I}}_1$ with $x_j(1)$ and $a_j(1)$. Interface $\mathscr{I}$ receives an output and view for α_1 which it denotes $x_j(2)$ and $a_j(2)$. Allowing r to range from 2 to $i-1$, $\mathscr{I}$ repeats this process with $\hat{\mathscr{I}}_r$, supplying $\hat{\mathscr{I}}_r$ with $a_j^\alpha(r)$ and receiving $a_j^\alpha(r+1)$. Finally, $\mathscr{I}$ supplies $\mathscr{I}_i$ with $a_j^\alpha(i)$ along with the $x_j^\alpha(i)$ and $v_j^{\beta_i}$ obtained from protocol β_i. A specification for $\mathscr{I}$ is given in Fig. 5.

Now, let us consider the ensemble $R_s(\vec{x}, \vec{a} \cdot a_A, k)$ induced by running s protocols

Interface $\mathscr{I}$

1 **For** $i = 1..f(k)$ **for** $j = 1..n$ **do** $x_j(i) \leftarrow \Lambda$, $a_j(i) \leftarrow \Lambda$.
2 **For** $i = 1..f(k)$ **do**
2.1 Participate in β_i using $\mathscr{I}_i$.
2.2 On request j from $\mathscr{I}_i$ **do**:
2.2.1 Corrupt j, obtain $v_j^{\beta_i}$, $x_j^{\beta}(i)$, $a_j^{\beta}(i)$.
2.2.2 Assign $x_j(1)$ from $x_j^{\beta}(i)$ and $a_j(1)$ from $a_j^{\beta}(i)$.
2.2.3 **If** $a_j(i) = \Lambda$ **then**
for $r = 1..(i-1)$ **do**
$q_{\hat{i}_r}, x_j(r+1), a_j(r+1) \leftarrow \hat{\mathscr{I}}_r(q_{\hat{i}_r}, x_j(r), a_j(r))$
2.2.4 Return $x_j(i) \cdot a_j(i) \cdot v_j^{\beta_i}$ to $\mathscr{I}_i$.
2.3 Initialize $\hat{\mathscr{I}}_i$ (start it on input $q_A = q_A(i)$) and record state q_{I_i}.

Fig. 5. Interface for the concatenation of $f(k)$ protocols from $\{\beta_i\}$.

from $\{\beta_i\}$ followed by $f(k) - s$ protocols from $\{\alpha_i\}$, with adversary $\mathscr{A}$ and a modified interface $\mathscr{I}_s$. Let $x_j(1) = x_j$ and $a_j(1) = a_j$ for $1 \le j \le n$ and set $a_A(1) = a_A$. The outputs of β_i for $i = 1$ to s are denoted $x_j^{\beta}(i+1)$, $a_j^{\beta}(i+1)$, $a_A^{\beta}(i+1)$, and the outputs of α_i for $i = s+1$ to $f(k)$ are denoted $x_j^{\alpha}(i+1)$, $a_j^{\alpha}(i+1)$, $a_A^{\alpha}(i+1)$. The modified interface $\mathscr{I}_s$ uses postprotocol corruption to create views for the first s β-protocols, but uses portions of the actual views (a_j^{α}) to construct views of the α-protocols. That is, $\mathscr{I}_s$ runs the $\mathscr{I}$ program as in Fig. 5, but it performs postprotocol corruption (step (2.2.3)) for r ranging only from 1 to $\min(i-1, s)$. If $i > s$, $\mathscr{I}_s$, then sets $x_j(i) \leftarrow x_j^{\alpha_i}(i)$ and, letting $A_j^{\alpha}(s, i)$ denote the suffix of $a_j^{\alpha}(i)$ corresponding to subprotocols $\alpha_{s+1}, \ldots, \alpha_i$, $\mathscr{I}_s$ sets $a_j(i) \leftarrow a_j(s) \cdot A_j^{\alpha}(s, i)$. The views $\mathscr{I}_s$ supplies are thus a combination of β and α views. We may write $R_s = [\hat{\alpha}(s+1) \circ \hat{\beta}(s), \mathscr{I}_s(\mathscr{A})]$, where $\hat{\alpha}(s) = \alpha_{f(k)} \circ \cdots \circ \alpha_s$ and $\hat{\beta}(s) = \beta_s \circ \cdots \circ \beta_1$.

Clearly, $\mathscr{I}_{f(k)}$ is $\mathscr{I}$ while $\mathscr{I}_0$ is no interface at all. Therefore, $R_{f(k)}(\vec{x}, \vec{a} \cdot a_A, k) = [\beta^f, \mathscr{I}(\mathscr{A})](\vec{x}, \vec{a} \cdot a_A, k)$ and $R_0(\vec{x}, \vec{a} \cdot a_A, k) = [\alpha^f, \mathscr{A}](\vec{x}, \vec{a} \cdot a_A, k)$. As in the proof of Lemma 3, a difference exceeding $\delta(k)$ for $k \ge k_0$ between $[\alpha^f, \mathscr{A}]$ and $[\beta^f, \mathscr{I}(\mathscr{A})]$ translates into a $\delta_{i+1}(k)$ difference between R_i and R_{i+1} for some i and $\vec{x} \cdot \vec{a} \cdot a_A$.

Expressing $\Pr_{R_i(\vec{x} \cdot \vec{a} \cdot a_A, k)}[\vec{y} \cdot \vec{v} \cdot y_A]$ as

$$\sum_{\vec{x}' \cdot \vec{a}' \cdot a_A'} \Pr_{[\hat{\alpha}(i+1),\, \mathscr{I}_i(\mathscr{A})](\vec{x}' \cdot \vec{a}' \cdot a_A', k)}[\vec{y} \cdot \vec{v} \cdot y_A] \cdot \Pr_{[\hat{\beta}(i),\, \mathscr{I}_i(\mathscr{A})](\vec{x} \cdot \vec{a} \cdot a_A, k)}[\vec{x}' \cdot \vec{a}' \cdot a_A'],$$

then as in the proof of Lemma 1, there exists a $\vec{x}' \cdot \vec{a}' \cdot a_A'$ such that

$$\sum_{\vec{y} \cdot \vec{v} \cdot y_A} |\Pr_{[\hat{\alpha}(i+1),\, \mathscr{I}_i(\mathscr{A})](\vec{x}' \cdot \vec{a}' \cdot a_A', k)}[\vec{y} \cdot \vec{v} \cdot y_A] - \Pr_{[\hat{\alpha}(i+2) \circ \beta_{i+1},\, \mathscr{I}_{i+1}(\mathscr{A})](\vec{x}' \cdot \vec{a}' \cdot a_A', k)}[\vec{y} \cdot \vec{v} \cdot y_A]| > \delta_{i+1}(k). \tag{2}$$

If $i < f(k)$, the difference of $\delta_{i+1}(k)$ between running α_{i+1} followed by $\alpha_{i+2}, \ldots, \alpha_{f(k)}$ and running β_{i+1} followed by $\alpha_{i+2}, \ldots, \alpha_{f(k)}$ will contradict the postprotocol corruptibility of α_{i+1} with respect to β_{i+1}. Consider protocols α_{i+1} and β_{i+1} executed on inputs $\vec{x}' \cdot \vec{a}' \cdot a_A'$. When $\mathscr{A}$ attacks α_{i+1}, it resets itself to the state listed in a_A' and continues where it seemingly left off. Let $\hat{\mathscr{A}}$ be a postprotocol adversary against α_i that, given $\vec{y} \cdot y_A$, runs $\mathscr{A}$ starting in the state described by y_A against an internally

simulated $\hat{\alpha}(i+2)$ on inputs $\vec{x}(i+2) = \vec{y}(i)$. Then (2) translates to the following:

$$\sum_{\vec{y}\cdot\vec{v}\cdot y_A} |\Pr_{[\alpha_{i+1}, \mathscr{A}, \mathscr{A}](\vec{x}'\cdot\vec{a}'\cdot a'_A, k)}[\vec{y}\cdot\vec{v}\cdot y_A] - \Pr_{[\beta_{i+1}, \mathscr{S}_{i+1}(\mathscr{A}), \hat{\mathscr{S}}_{i+1}(\hat{\mathscr{A}})](\vec{x}'\cdot\vec{a}'\cdot a'_A, k)}[\vec{y}\cdot\vec{v}\cdot y_A]| > \delta_{i+1}(k)$$

for some $k \geq k_0$, contradicting that α_{i+1} is postprotocol corruptible with respect to β_{i+1}.

If $i = f(k)$, then (2) says that there exists a $\vec{x}'\cdot\vec{a}'\cdot a'_A$ such that

$$\sum_{\vec{y}\cdot\vec{v}\cdot y_A} |\Pr_{[\alpha_{f(k)}, \mathscr{A}](\vec{x}'\cdot\vec{a}'\cdot a'_A, k)}[\vec{y}\cdot\vec{v}\cdot y_A] - \Pr_{[\beta_{f(k)}, \mathscr{S}_{f(k)}(\mathscr{A})](\vec{x}'\cdot\vec{a}'\cdot a'_A, k)}[\vec{y}\cdot\vec{v}\cdot y_A]| > \delta_{f(k)}(k)$$

for some $k \geq k_0$, showing $\alpha_{f(k)} \not\approx^{\delta_{f(k)}(k)} \beta_{f(k)}$, again a contradiction. □

The following states an oft-used but previously unproven and unformalized folk theorem:

Theorem 5. *Concatenating polynomially many postprotocol-corruptible protocols with identical convergence parameters preserves resilience*:

$$\{\alpha_i\} \trianglerighteq \{\beta_i\} \quad \Rightarrow \quad \alpha^f \geq \beta^f,$$

where f is polynomially bounded. This holds for exponential and statistical resilience.

Proof. The result follows directly from Lemma 4. □

An essential technical point to note is that *uniform* pairwise resilience is required; simply composing polynomially many ensembles or protocols will not preserve indistinguishability or security. The pitfall is analogous to the distinction between functions that converge pointwise and functions that converge uniformly. Consider, for example, the ensembles P_i and Q_i defined by

$$\Pr_{P_i(z,k)}[0] = 1 \qquad \text{if} \quad k < 2i,$$

$$\Pr_{P_i(z,k)}[1] = 1 \qquad \text{if} \quad k \geq 2i,$$

$$\Pr_{Q_i(z,k)}[1] = 1 \qquad \text{always.}$$

Clearly, for all i, $P_i \approx^{O(2^{-k})} Q_i$, since $(\forall i, z)\ (\forall k \geq 2i)\ P_i(z,k) = Q_i(z,k)$. Letting $f(k) = k$, we see $(\forall z, k)\ \Pr_{P^f(z,k)}[0] = 1$, because $P_{f(k)}(z,k) = P_k(P_{k-1}(\cdots), k)$ and $k < 2k$. On the other hand, $(\forall z, k)\ \Pr_{Q^f(z,k)}[0] = 0$. Hence $(\forall z, k)\ |P^f(z,k) - Q^f(z,k)| = 1$, eliminating the possibility that $P^f \approx^{O(f(k)\cdot 2^{-k})} Q^f$. The conclusion of Lemma 3 fails without the uniformity condition. Considering protocols, let $F(x_1, \ldots, x_n) = 0$ always. In protocol α_i, player 1 reveals x_1 when $k < 2i$; no other messages are ever sent and every player always outputs 0. In protocol β_i, no messages are ever sent and every player always outputs 0. Clearly, $\alpha_i \geq^{O(2^{-k})} \beta_i$, but for $f(k) = k$, α^f *always* reveals x_1 while β^f reveals *nothing*. It easily follows that $\alpha^f \not\geq^{O(f(k)\cdot 2^{-k})} \beta^f$.

4.5. *Concatenating Ideal Protocols*

The direct concatenation of ideal protocols does not quite provide the desired level of resilience we expect. Operating two protocols $\mathsf{ID}(F)$ and $\mathsf{ID}(G)$ in sequence does

not ensure that the inputs to the second protocol are in any way related to the inputs or outputs of the first. A corrupt player might obtain $y_i(1)$ as its output of F but instead supply $y_i(1) + 7$ or 2 or $x_i(1) - 63$ as its input $x_i(2)$ to G.

In many circumstances this is quite undesirable. For example, the employees of a company might wish to calculate their average overall salary and the average salary of management. First, they compute the overall average. For the second computation, however, the management may report lower salaries to obtain an advantage in salary negotiations.

Let $\mathcal{F} = \{F^r\}_{r \in \mathbf{N}}$ be a possibly infinite collection of functions $F^r \in \mathrm{PFF}$, and let $f: \mathbf{N}^3 \to \mathbf{N}$ be polynomially bounded.

Definition 20. The *open concatenation of f ideal protocols* from $\mathcal{F}$ is the following protocol, denoted by $\circ\mathsf{ID}(F^i)$. Consider $n + f(n, m, k)$ players. Players $i > n$ are incorruptible. The protocol requires $2f(n, m, k)$ rounds; let r range from 2 to $f(n, m, k)$:

(*Round* 1) Each player $i \in [n]$ sends $x_i(1) = x_i$ to trusted player $(n + 1)$.

(*Round* 2) Player $(n + 1)$ computes $F^1(x_1(1), \ldots, x_n(1))$ and returns $y_i(1)$ to each player $i \in [n]$.

(*Round* $2r - 1$) Each player $i \in [n]$ sends $x_i(r) = y_i(r - 1)$ to trusted player $(n + r)$.

(*Round* $2r$) Player $(n + r)$ computes $F^r(x_1(r), \ldots, x_n(r))$ and returns $y_i(r)$ to each player $i \in [n]$.

Technically, the objection simply to concatenating protocols directly is the following. Operating ideal protocols in sequence invokes *different* trusted parties, one to compute F^1, one to compute F^2, and so on. None of the trusted parties share any information or communicate at all. In the ideal *composite* protocol, the host retains the values of the outputs for use in each successive computation.

Definition 21. The *ideal composite protocol* $\mathsf{IDC}(\circ F^i)$ has *one* trusted party. The trusted host requests $x_1(1), \ldots, x_n(1)$, computes F^1, computes F^2 on the results, and so on, up to the computation of $F^{f(n,m,k)}$. It returns the values $y_i(1), \ldots, y_i(f(n, m, k))$ to each player i. (Notation: every appearance of "$\circ$" within $\mathsf{IDC}(\cdot)$ indicates a function value to return.)

Robustness, privacy, and threshold schemes (see Section 3.3) resolve the problem of inconsistent arguments being used in different stages of a concatenation of protocols. Robust computations are performed equally well when several different trusted hosts each compute an individual stage in the computation, as when a single host computes and returns all the intermediate results (ensuring outputs of earlier functions are used as inputs to later ones). Computing and revealing sequence of private functions is as secure as computing only the final result. These properties come together in the following theorem, which demonstrates that the share/add/multiply/reveal structure found so often in multiparty protocol design—though not proven—is as resilient as having a single trusted host compute and return only the final result. (In fact, to the author's knowledge, only one protocol departs from this

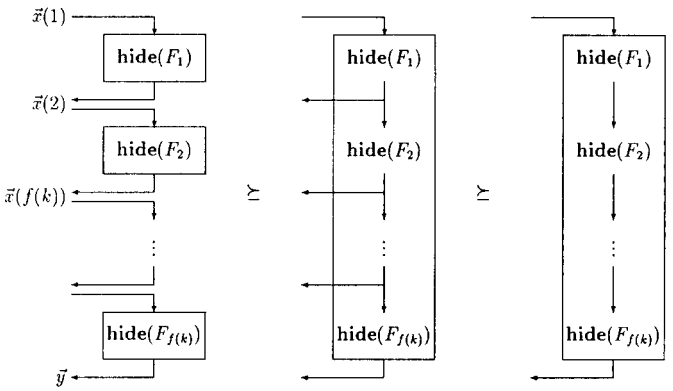

Fig. 6. Illustration of ideal protocols for **hide**(F^f). The first protocol, $\circ_i$ ID(**hide**(F^i)), contains several hosts who compute robust and private representations of intermediate results. The second protocol, IDC($\circ_i$ **hide**(F^i)), contains just one host who computes and returns the same values. The third protocol, ID($\circ_i$ **hide**(F^i)) (equivalently ID(**hide**(F^f))), contains just one host who computes and returns only the final representation.

paradigm: the constant-rounds noncryptographic protocol of Bar-Ilan and Beaver [2], which uses an even more general technique to reduce the number of rounds of interaction. See [6] for a formal statement of this more general methodology.) Define $F^f = F^{f(n,m,k)} \circ \cdots \circ F^1$, and for any function H, let **hide**(H) be the function **sha** $\circ$ H $\circ$ **rec**.

Theorem 6. *If* (SHA, REC) *is a t-threshold scheme computing* (**sha**, **rec**), *then*

$$\text{REC} \circ (\circ \text{ID}(\textbf{hide}(F^i))) \circ \text{SHA} \succeq \text{ID}(F^f).$$

Proof. The proof contains four stages, some of which are illustrated in Fig. 6. The first stage uses only the property of *robustness* to show that, even though individual trusted hosts do not communicate among themselves, a sequence of hosts is as resilient as a single host (who guarantees that the outputs of F^i are used as inputs to F^{i+1}). The problem of faulty players who substitute false values for the outputs of earlier protocols is thus solved. Formally, if **sha** is t-robust, then, for any family of functions $\{F^i\}$,

$$\circ_i \text{ID}(\textbf{hide}(F^i)) \succeq \text{IDC}(\circ_i \textbf{hide}(F^i)). \tag{3}$$

Let $\alpha = \circ_i \text{ID}(\textbf{hide}(F^i))$ and $\beta = \text{IDC}(\circ_i \textbf{hide}(F^i))$. In potocol α, there are $f(n, m, k)$ trusted hosts: host $(n + i)$ computes **hide**(F^i). Inductively, by robustness, as long as no more than t values of the outputs of **hide**(F^i) are changed, then the output of F^{i+1} is $F^{i+1} \circ \cdots \circ F^1(x_1, \ldots, x_n)$. Therefore, the outputs of players in α and β—namely, just the list of $f(n, m, k)$ values returned by the hosts—are identical, if the x_i values given to host $(n + 1)$ in α are the same as those given to $(n + 1)$ in β.

At the start of α, interface $\mathcal{I}$ responds to requests to corrupt player i by requesting that i be corrupted in β. It then returns x_i and a_i to $\mathcal{A}$. When $\mathcal{A}$ outputs its substituted messages from corrupted players to host $(n + 1)$ in α, $\mathcal{I}$ simply repeats these messages—which represent alternatively chosen x_i' inputs—to the single host

$(n + 1)$ in β. In both protocols, the original set of inputs is easily seen to be the same. Hence the computations of F^i are identical. Now, $\mathcal{I}$ completes protocol β, obtaining messages $y_i(1), \ldots, y_i(f(n, m, k))$ for each corrupted player. In α, these computations have not yet occurred, but now $\mathcal{I}$ is ready to supply them to $\mathcal{A}$ when needed. Furthermore, when $\mathcal{A}$ requests a new corruption, $\mathcal{I}$ simply requests that player i be corrupted in β, obtaining the entire sequence of returned values, which $\mathcal{I}$ then uses to supply $\mathcal{A}$ gradually with the results returned by the sequence of hosts in α. This completes the proof of (3).

The second stage uses the property of *privacy* to show that a protocol in which a single host reveals private representations of intermediate results is as resilient as one in which the single host reveals only the final representation:

$$\mathsf{IDC}(\circ_i \mathbf{hide}(F^i)) \succeq \mathsf{ID}(\circ_i \mathbf{hide}(F^i)). \tag{4}$$

Let $\alpha = \mathsf{IDC}(\circ_i \mathbf{hide}(F^i))$ and $\beta = \mathsf{ID}(\circ_i \mathbf{hide}(F^i))$. Both protocols contain a single trusted host who returns a result (or string of results) in one step. Because there is only one host in each case, the outputs of nonfaulty players are identical whenever the inputs are the same in both protocols. Interface $\mathcal{I}$ makes sure that requests by $\mathcal{A}$ to replace inputs are sent to protocol β to ensure that the inputs are the same.

The only difference between protocols is that in α the view of a corrupted player includes several progressive outputs from the computations of $\circ_1^j \mathbf{hide}(F^i)$. Because **sha** is t-private, an ideal protocol to compute the output $\mathbf{hide}(F^i) = \mathbf{sha} \circ F^i \circ \mathbf{rec}$ at each step is as private as the ideal vacuous protocol. Hence there exists a subinterface $\mathcal{I}_{\mathbf{sha}}$ that $\mathcal{I}$ can run to take care of the computation. Because $\mathcal{I}_{\mathbf{sha}}$ itself expects to participate in a vacuous protocol, it expects only initial inputs $x_i \# a_i$ in response to its corruption requests. Interface $\mathcal{I}$ generates the intermediate inputs by postprotocol corruption of earlier incarnations of $\mathcal{I}_{\mathbf{sha}}$. Thus $\mathcal{I}$ is able to provide the proper responses to $\mathcal{A}$, simulating the vacuous protocol without using additional information.

The third stage is to show that if (SHA, REC) is a t-threshold scheme for functions (**sha**, **rec**), then, for any function H,

$$\mathsf{ID}(\mathbf{rec}) \circ \mathsf{ID}(\mathbf{hide}(H)) \circ \mathsf{ID}(\mathbf{sha}) \succeq \mathsf{ID}(H). \tag{5}$$

Let $\alpha = \mathsf{ID}(\mathbf{rec}) \circ \mathsf{ID}(\mathbf{hide}(H)) \circ \mathsf{ID}(\mathbf{sha})$ and let $\beta = \mathsf{ID}(H)$. Robustness implies that, provided inputs are the same in α and β, the outputs are the same, since $\mathbf{rec} \circ \mathbf{hide}(h) \circ \mathbf{sha} = H$. As in the second stage above, protocol α reveals intermediate values that are private representations, and the previous arguments hold. When $\mathcal{A}$ requests the *final* view and output of $\mathsf{ID}(\mathbf{rec})$ for a corrupt player, $\mathcal{I}$ supplies $\mathcal{A}$ with the value lifted directly from corrupting that player in $\mathsf{ID}(H)$.

The fourth and final stage puts these pieces together to prove the theorem. Using (3) and (4),

$$\begin{aligned} \circ_i \mathsf{ID}(\mathbf{hide}(F^i)) &\succeq \mathsf{IDC}(\circ \mathbf{hide}(F^i)) \\ &\succeq \mathsf{ID}(\mathbf{hide}(F^f)). \end{aligned}$$

Using Theorem 5 and result (5),

$$\begin{aligned} \mathrm{REC} \circ (\circ \mathsf{ID}(\mathbf{hide}(F^i))) \circ \mathrm{SHA} &\succeq \mathrm{REC} \circ \mathsf{ID}(\mathbf{hide}(F^f)) \circ \mathrm{SHA} \\ &\succeq \mathsf{ID}(F^f). \end{aligned}$$

□

4.6. *Security with High Probability*

To achieve an exponentially resilient protocol, it suffices to give a protocol that provides security for most runs. Let **Defect** be a predicate on strings that states whether an execution producing $\vec{y} \cdot \vec{v} \cdot y_A$ is "defective." We leave the particular definition of a defective run open for the moment, but, as an example, it may label as defective a run in which a faulty player gives a successful proof of a false statement to a nonfaulty player, or one in which a faulty player shares a secret that the players accept after a verification procedure even though it cannot later be reconstructed. In particular, we consider predicates that are computable strictly from the adversary's output (the examples just given are such). It is also useful to allow the adversary's output to be a particular symbol, "*defect*," such that **Defect**("*defect*") = 1; an interface may output "*defect*" when it comes to a point where the adversary "gets away with" such cheating. In such a situation, the interface would *not* have the information to supply or the influence on the β protocol that it needs to match exactly what the adversary would gain in the α protocol.

Define an ensemble induced by nondefective executions:

$$[[\Pi, \mathscr{A}]](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k)$$
$$= \{Q \leftarrow \mathbf{ExecA}(n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k) \colon Y(Q) | (\mathbf{Defect}(Q) = 0)\}.$$

The term *optimistic* connotes ensembles conditioned on nondefective runs. Note that this definition applies equally well to any protocol, since **Defect** is defined on strings.

One protocol is optimistically as resilient as another if there is an interface that works when the runs are not defective:

Definition 22. A protocol α is *optimistically* as resilient as protocol β with respect to adversary classes A_α and A_β, if there exists an interface $\mathscr{I}$ from α to β such that, for all adversaries $\mathscr{A} \in \mathsf{A}_\alpha$,

$$[[\alpha, \mathscr{A}]] \approx [[\beta, \mathscr{I}(\mathscr{A})]].$$

We write this as

$$\alpha \dot{\geq} \beta.$$

Let the maximal probability of a defective run in protocol α with adversary class A_α be

$$\mathbf{PrDefect}(\alpha) = \mathbf{max}_{(\mathscr{A} \in \mathsf{A}_\alpha, n, m, \vec{x}, \vec{a}, a_A)} \Pr[\mathbf{Defect}(\mathbf{ExecA}(n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k)) = 1]$$

as a function of k. Similarly define $\mathbf{PrDefect}(\beta)$. In order to show a protocol resilient, it suffices to prove that it is optimistically resilient, as long as defective runs are rare:

Theorem 7. *If* $\alpha \dot{\geq}^{\mathrm{e}} \beta$ *and, for some* $c > 1$, $\mathbf{PrDefect}(\alpha) = O(c^{-k})$ *and* $\mathbf{PrDefect}(\beta) = O(c^{-k})$, *then* $\alpha \geq^{\mathrm{e}} \beta$.

Proof. Note first that various combinations of perfect, exponential, and statistical resilience are also possible. By the premise of the theorem, there is an interface and

a constant d satisfying $[[\alpha, \mathscr{A}]] \approx^{d^{-k}} [[\beta, \mathscr{I}(\mathscr{A})]]$. We use the following elementary lemma:

Lemma 8. *Let P be a distribution on X and let $Y \subseteq X$ be such that $\Pr[Y] \geq 1 - \delta$ for some $\delta \leq \frac{1}{2}$. Then $|P - (P|Y)| \leq 3\delta$.*

Proof. Since $\delta \leq \frac{1}{2}$, we have $1/(1-\delta) \geq 1 + 2\delta$, and

$$\begin{aligned}\sum_{x \in Y} |\Pr[x|Y] - \Pr[x]| &= \sum_{x \in Y} \left(\frac{\Pr[x]}{\Pr[Y]} - \Pr[x] \right) \\ &= \left(\frac{1}{1-\delta} - 1 \right) \cdot \Pr[Y] \\ &\leq (1 + 2\delta - 1)(1 - \delta) \leq 2\delta - 2\delta^2.\end{aligned}$$

Now consider the points not in Y:

$$\sum_{x \notin Y} |\Pr[x|Y] - \Pr[x]| = \sum_{x \notin Y} \Pr[x] = \delta.$$

Therefore,

$$\sum_{x \in X} |\Pr[x|Y] - \Pr[x]| \leq 3\delta - 2\delta^2 \leq 3\delta. \qquad \square$$

Because $\mathbf{PrDefect}(\alpha) = O(c^{-k})$ and $\mathbf{PrDefect}(\beta) = O(c^{-k})$, there exist k_α and k_β such that $k \geq k_\alpha \Rightarrow \mathbf{PrDefect}(\alpha) \leq c^{-k}$ and $k \geq k_\beta \Rightarrow \mathbf{PrDefect}(\beta) \leq c^{-k}$; we may assume $c^{-k} \leq \frac{1}{2}$ without loss of generality. In particular, setting $\delta = \delta(k) = c^{-k}$ and letting Y be the set of defective runs for α, Lemma 8 implies that, for all for $k \geq k_\alpha$, for all adversaries $\mathscr{A} \in \mathsf{A}_\alpha$, and for all $n \# m \# \vec{x} \cdot \vec{a} \cdot a_A$,

$$|[\alpha, \mathscr{A}](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k) - [[\alpha, \mathscr{A}]](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k)| \leq 3c^{-k}.$$

Likewise, for $k \geq k_\beta$, and noting additionally that $\mathscr{A} \in \mathsf{A}_\alpha$ implies $\mathscr{I}(\mathscr{A}) \in \mathsf{A}_\beta$, we have

$$|[[\beta, \mathscr{I}(\mathscr{A})]](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k) - [\beta, \mathscr{I}(\mathscr{A})](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k)| \leq 3c^{-k}.$$

Set $k_0 = k_\alpha + k_\beta$; clearly, for $k \geq k_0$, the differences in probability distributions are not large:

$$\begin{aligned}&|[\alpha, \mathscr{A}](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k) - [[\alpha, \mathscr{A}]](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k)| \\ &\qquad + |[[\alpha, \mathscr{A}]](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k) - [[\beta, \mathscr{I}(\mathscr{A})]](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k)| \\ &\qquad + |[[[\beta, \mathscr{I}(\mathscr{A})]]](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k) - [\beta, \mathscr{I}(\mathscr{A})](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k)| \\ &\quad \leq 3c^{-k} + d^{-k} + 3c^{-k}.\end{aligned}$$

Letting $C = 1/7 \cdot \min\{c, d\}$ and using the triangle inequality, the following holds for all $\mathscr{A} \in \mathsf{A}_\alpha$ and for all $n \# m \# \vec{x} \cdot \vec{a} \cdot a_A$:

$$k \geq k_0 \quad \Rightarrow \quad |[\alpha, \mathscr{A}](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k) - [\beta, \mathscr{I}(\mathscr{A})](n \# m \# \vec{x} \cdot \vec{a} \cdot a_A, k)| \leq C^{-k},$$

completing the proof of the theorem. $\square$

4.7. *The Strong Principle of Independence*

Another oft-cited but imprecise principle is that "the messages the adversary sees from nonfaulty players are independent of their inputs, so the protocol is secure." In fact, it is often the case that nonfaulty behavior is not completely independent of the inputs—for example, when defective runs are possible. Independence by itself says nothing directly, for example, about an adversary's *influence* or about computationally expensive messages (providing free, uncomputable data in a resource-bounded setting). Message independence might not hold when comparing two slightly insecure protocols and is thus not always applicable to proving relative resilience. A precise formulation, proof, and description of the applicability of this principle is required.

The principle of message independence reflects a more fundamental statement: optimistically, the messages from nonfaulty players in α are a fixed probabilistic function of adversarial outputs in α and messages seen by corrupted players in β. When $\beta = \mathrm{ID}(F)$, the messages in β contain only the final output, so by fiat, the nonfaulty messages in α are independent of all information except the output. The adversary's computations are based on these nonfaulty messages and on inputs of faulty players, and as a result they are inductively independent of nonfaulty players' information.

Definition 23. *The Strong Principle of Independence* for α and β: Optimistically (i.e., conditioned on nondefective runs), **Fault** is a function of $\mu(T, [n], 1..r)$ and $\mu([n], T, 1..r)$ in each round r of protocol α and of $\mu^{\beta}(T, [n], 1..R^{\beta})$ and $\mu^{\beta}([n], T, 1..R^{\beta})$ in protocol β. As in Definition 7, T indicates the coalition chosen by the adversary at round r of α; R^{β} is the number of rounds in protocol β.

Theorem 9. *If the strong principle of independence holds for α and β, and* **PrDefect** $= O(\delta(k))$, *then* $\alpha \geq^{O(\delta(k))} \beta$.

Proof. A canonical interface $\mathscr{I}$ operates internal versions of nonfaulty players in α, each starting with input $x_i = 0$ (or some allowed input). Intuitively, it should not matter with respect to messages sent to $\mathscr{A}$ whether nonfaulty player i holds 0 or some other input. When the interface must evaluate $\mathbf{Fault}(i, \langle \vec{q}, \mu^{\mathrm{in}}, v, \vec{q}^{\,\mathrm{new}}, \mu^{\mathrm{out}}, \mu^{\mathrm{del}} \rangle)$ in response to a new corruption request i from $\mathscr{A}$, it requests that i be corrupted in β. It then restarts the *internal* set of players, this time with x_i set to the value obtained from β. At each step of the internal execution, $\mathscr{I}$ makes probabilistic calculations for each internal player as specified by α, but conditioned on messages already sent to and received from $\mathscr{A}$ (i.e., $\mu(T, [n], 1..r)$ and $\mu([n], T, 1..r)$) and on messages sent to and received from $\mathscr{I}$ in β.

For example, if player i has secretly shared x_i, then $\mathscr{I}$ repeats the sharing with the condition that $\mathrm{PIECE}_j(x_i)$ for $j \in T$ is unchanged; in this case, the conditional distribution on remaining pieces is efficiently calculable, simply by selecting a certain number of remaining pieces uniformly at random and solving linear equations to determine the rest. Care must be exercised: if a secret s from player i has already been reconstructed, then later messages seen by $\mathscr{A}$ contain all pieces, so no

new selection of pieces is required. These checks are normally simple. In the case of Turing machines, appropriate areas of player i's random tape must be overwritten to comply with messages committed to $\mathscr{A}$. This, too, is normally easy to compute.

The Strong Principle of Independence allows a canonical interface to be successful. Because **Fault** is expressible as a function solely of messages to and from $\mathscr{A}$ in α (here, actually to and from $\mathscr{I}$) and messages to and from $\mathscr{I}$ in β, all these messages have nonzero probability for any set of inputs $\vec{x}$, hence the conditional distributions are well defined. Note that there is no guarantee that the conditional distributions are simple, but normally they are, being the result of a uniform bit selection or a linear algebraic calculation. Note also that in the *complexity-based* setting, an interface may easily fail at this point because the messages it has committed to $\mathscr{A}$ are not consistent with *any* execution, so the conditional distributions are ill defined and impossible to satisfy. (A standard trick in complexity-based settings is to "simulate" an encrypted value for which a decryption is not known by replacing it with a random string. This may lead to problems when a dynamic adversary later corrupts the sender, becoming able to decode earlier messages. Claims of security against dynamic adversaries (for example, those made in [20] and [21]) fall on this issue, unless other techniques such as tape-erasing in a memoryless model are employed [11].) In the information-theoretic setting, such issues fail to arise (and the presence of private channels obviates the need to deal with computationally complex functions).

Thus, a canonical interface can, in the absence of defective runs, produce the necessary conditional computations and views that match *exactly* those in an execution of $[\alpha, \mathscr{A}]$. The output of $\mathscr{I}$, after it repeats the internal simulation and extracts the view of player i, is identical to the output of **Fault** in $[\alpha, \mathscr{A}]$. Hence, optimistically, $\alpha \gtrsim \beta$. By Theorem 7, $\alpha \gtrsim^{\delta(k)} \beta$. □

5. Multiparty Protocols with Faulty Minority

Ben-Or *et al.* [14] and Chaum *et al.* [15] present protocols secure against passive t-adversaries for $t < n/2$. The protocol presented in this paper follows [14], with modifications: a modified version of Rabin's solution to verifiable secret sharing for $t < n/2$ is employed, and a new method to protect against $t < n/2$ faults during secret multiplication is presented. The latter constitutes the major algorithmic contribution of this paper. Previously, solutions existed only for $t < n/3$.

The overall structure of the main protocol EVAL to compute $F \in$ PFF is simple: secretly share inputs; secretly recombine shared values to compute a new secret equal to $F(x_1, \ldots, x_n)$; reconstruct the final secret, $F(x_1, \ldots, x_n)$.

Let C_F be an arithmetic circuit for F over finite field $E = \mathrm{GF}(2^{nm})$. For ease of protocol description, assume without loss of generality that $C_F(m, n)$ is an array of gates of width $W(m, n)$ and depth $D(m, n)$, with multiplicative gates on even levels and additive gates on odd levels. The values at level l of the circuit are denoted $x_{l1}, x_{l2}, \ldots, x_{lW}$. Each gate $g(l, i)$ is represented by a string of the form $\langle \Sigma, c_0, \ldots, c_W \rangle$ indicating a computation $x_{li} \leftarrow c_0 + \sum_{j=1}^{W} c_j x_{(l-1)j}$, or by a string $\langle \Pi, j, k \rangle$ indicating $x_{li} \leftarrow x_{(l-1)j} x_{(l-1)k}$. The values at level 0 are simply the inputs:

UNIVGATE $(C_F, l, i, x_{(l-1)1}, \ldots, x_{(l-1)W}, x_{li})$

if $g(l, i) = \langle \Sigma, c_0, \ldots, c_W \rangle$ **then**
  LINEAR-COMBINE$(g(l, i), x_{(l-1)1}, \ldots, x_{(l-1)W}, x_{li})$
else if $g(l, i) = \langle \Pi, j, k \rangle$ **then**
  MULTIPLY$(g(l, i), x_{(l-1)1}, \ldots, x_{(l-1)W}, x_{li})$

Fig. 7. Universal gate protocol to add or multiply secrets, producing secret output x_{li}.

$x_{0i} = x_i$ for $1 \leq i \leq n$ and $x_{0i} = 0$ for $n < i \leq W$. No wire skips a level; an additive gate with all constants 0 except one supports this assumption without loss of generality. The circuit is evalutated level by level, evaluating $g(l, 1)$ to $g(l, W)$ at each level.

Without loss of generality, the output of F is described by the entire output layer, which all players are allowed to see. This assumption is made for the sake of presentability. The output values can be divided into subsets seen by each player, if desired. It is also worth noting that if each player supplies a sequence of random bits and the circuit uses these bits to mask portions of the output, then the entire output may be revealed without loss of security, since each player can effectively read only the portion whose mask it knows.

Once values have been secretly shared, the EVAL protocol consists merely of applying a universal gate protocol UNIVGATE to create new secrets as functions of the previous level. The universal gate protocol runs an addition protocol or a multiplication protocol, depending on $g(l, i)$ (Fig. 7). These protocols, LINEAR-COMBINE and MULTIPLY, are described in later sections.

The term "completeness" has been used to describe techniques for general multiparty computation. A formal statement of the general methodology for "completeness" results follows:

Theorem 10 ("Completeness" Paradigm). *Let* (SHA, REC) *be a t-threshold scheme. Let* $F \in$ PFF *be such that there exists a function family* $\{F^i\}$ *such that, for some polynomially bounded* $f(n, m, k)$, $F = \circ_1^{f(n,m,k)} F^i$. *If there exists a family* $\{\alpha_i\}$ *of protocols that* uniformly *t-resiliently computes robust and secret representations* (*e.g., secretly shared*) *of* $\{F^i\}$, *then* REC $\circ$ α^f $\circ$ SHA (*where* $\alpha^f = \circ_1^{f(n,m,k)} \alpha_i$) *is a t-resilient protocol for F. This holds for perfect, exponential, and statistical resilience.*

Proof. By Theorem 5, $\alpha^f = \circ\alpha_i \succeq \circ$ ID(**hide**(F^i)) and REC $\circ$ α^f $\circ$ SHA $\succeq$ REC $\circ$ ($\circ$ID(**hide**(F^i))) $\circ$ SHA. By Theorem 6, REC $\circ$ ($\circ$ID(**hide**(F^i))) $\circ$ SHA $\succeq$ ID(F). The result follows by the transitivity of $\succeq$ (Theorem 2). Furthermore, each theorem in the proof applies to exponential and statistical resilience as well. □

Theorem 11 (Main Result). *For all* $F \in$ PFF *and for all* $t(n) < n/2$, *there exists an exponentially t-resilient protocol for F. The protocol has message complexity and round complexity polynomial in n, m, k.*

Proof. Let $F \in$ PFF be computed by arithmetic circuit family $C_F = \{C_F(n, m)\}$ having depth $D(n, m)$ and width $W(n, m)$. Protocol EVAL is described in Fig. 8. It

EVAL $(C_F, x_1, \ldots, x_n, x_{D1}, \ldots, x_{DW})$

1 $(1 \le i \le n)$ i: SHARE(x_i) (as input secret x_{0i})
2 **for** $l = 1..D(n, m)$ **do**
  for $j = 1..W(n, m)$ **do**
    UNIVGATE$(C_F, l, j, x_{(l-1)1}, \ldots, x_{(l-1)W}, x_{lj})$
3 **for** $j = 1..W(n, m)$ **do**
    RECONSTRUCT$(x_{D(n,m),j})$

Fig. 8. A t-resilient protocol to evaluate a function F at inputs $x_1, \ldots, x_n$. Implicit in each subprotocol is the recovery protocol RECOVER, which reconstructs secrets of players who fault, so that the computation may continue.

requires a t-threshold scheme (SHARE, RECONSTRUCT) and t-resilient subprotocols LINEAR-COMBINE and MULTIPLY. These protocols are described and proven resilient in later sections (see Sections 5.1–5.3). Protocols LINEAR-COMBINE and MULTIPLY compute robust and private representations of linear combinations (**hide(LinComb)**) and products (**hide**($\times$)). Because a finite number (two) of protocol types are used, relative resilience is *uniform*. Hence the conditions of Theorem 10 are met, so EVAL exponentially t-resiliently computes F. □

5.1. *Verifable Secret Sharing and Time Capsules*

The first tool required for protocol EVAL is a method for verifiable secret sharing (VSS). Shamir's technique (dealer selects $p(u) \leftarrow UPoly(t, s)$ and sends $\text{PIECE}_i(s) = p(i)$ to each player i) provides a t-threshold scheme against passive adversaries, but when adversaries may change values, two important properties fail:

1. After sharing, all nonfaulty players know whether the dealer is cheating.
2. The players hold a robust representation of s (i.e., a unique secret is reconstructible, regardless of faulty behavior).

Several schemes to satisfy the added requirements of *verifiability* have been proposed [16], [14], [15], [28], [29]. Rabin has proposed the first method tolerating $t < n/2$ in the noncryptographic model. Her approach follows Shamir's outline, but utilizes many three-party subprotocols to ensure that pieces are not altered (or are, at worst, omitted), and it utilizes a cut-and-choose method (based on [15]) to ensure that a dealer who does not select $p(u) \in \text{Poly}(n, t, s)$ for some s is disqualified.

The key tool to Rabin's method for VSS is a three-party protocol based on "check-vectors." This protocol solves a problem described below as the Time Capsule problem. An alternate solution requiring no field multiplications is presented here. Rabin's original method solved the first half of the problem of achieving multiparty protocols for $t < n/2$; a solution for the second half, based on a problem described later as the ABC Problem, is the main result of this paper.

5.1.1. *Time Capsules.* The Time Capsule problem is a kind of three-party secret-sharing scheme: a *sender* S passes a secret b to an *intermediary* I, who later passes it on to a *receiver*, R. The receiver R must be able to detect tampering (as long as S is nonfaulty,) and I must know if S provided valid information with which to convince R of the message. The ideal protocol ID(**tc**) for time-capsules is as follows:

in round 1, S sends b to the trusted host, and I and R send a 0 to the host. In round 2, the host returns b to I and R if both sent a message other than 0 (if both misbehave, then they *can* learn b). In the second phase, starting with round 3, I sends 0 or 1 to the host, who in round 4 sends b on to R if I sent a 1. (The intermediary can choose that the encapsulated information b is *not* passed on. Recall the implicit assumption that the host reports whether S or I cheats.)

Rabin's solution to this key problem uses linear polynomials over a finite field; S sends R a random line $l(x)$ and sends I the value x_0 at which $l(x_0) = b$. The intermediary cannot tamper with b without also guessing the line. The coefficients are called a "check vector," and they ensure detection of cheating with high probability.

An alternate method, derived from later work of the author with Feigenbaum and Shoup, is described in Fig. 9. A one-time pad (*pad*) known to S and R is used. This pad consists of two columns of $2k$ uniformly random field elements. The secret b is used to generate a mask (*mask*) which, added to the pad, gives a masked table (*cypher*). The mask corresponds to adding b to one or the other entry in each row, according to a uniformly random key (*key*) of $2k$ bits. The sender sends b and *cypher* to I and *key* and *pad* to R. Later, I sends *cypher* to R, who obtains b using *key* and *pad*. If at least $k/2$ rows are consistent with some value b, then R accepts; otherwise it declares CHEATING. In order to keep the secret hidden from I, b is actually split into two random bits whose exclusive-or is b; R uses one to decode the bit obtained from the process just described.

As in [28], a cut-and-choose method is used to ensure that I can later convince R of the value of b_I. The intermediary requests a random subset $\{i_1, \ldots, i_k\}$ of the

TC(S, I, R, b)

Phase I.		
1.1	S:	$b_R \leftarrow \text{unif}(E)$
		$b_I \leftarrow b_R + b$
		$key \leftarrow \text{unif}(\{0,1\}^{2k})$
		$pad(1..2k, 0..1) \leftarrow \text{unif}(E^{4k})$
	$(1 \le i \le 2k, 0 \le j \le 1)$	$mask(i,j) \leftarrow b_I \cdot (key(i) \oplus j)$
		$cypher \leftarrow mask + pad$
1.2	$S \rightarrow I$:	$\langle b_I, cypher \rangle$
1.3	$S \rightarrow R$:	$\langle b_R, key, pad \rangle$
2	$I \rightarrow R$:	$\langle i_1, \ldots, i_k \rangle \leftarrow uniform(\{S \subset 2^{[2k]} \mid \lvert S \rvert = k\})$
3	$R \rightarrow I$:	$\langle key(i_1), pad(i_1, 0..1), \ldots, key(i_k), pad(i_k, 0..1) \rangle$
4	I:	**if** inconsistent **then** broadcast CHEATING
Phase II.		
5.1	$I \rightarrow R$:	$\langle b_I, cypher \rangle$
5.2	R:	$decode \leftarrow cypher - pad$
		if $(\exists b_I)\,(\forall i, j)\,(decode(i,j) = b_I \cdot (key(i) + j)$
		then output $b_I - b_R$
		else output CHEATING

Fig. 9. Time capsule protocol for S to leave a message b with I, who later relays it to R. Tampering by I and misbehavior by S are checked. Addition of two tables indicates addition entry by entry. The protocol requires addition over some group, but not multiplication: thus exclusive-or (addition mod 2) may be used, or addition over a field E in order to be compatible with secret sharing.

rows of the key and pad from R. If any discrepancies occur then I broadcasts that it has detected cheating. If S has introduced enough discrepancies to cause a problem when I passes on b_i then I will detect it with very high probability.

Theorem 12. *Protocol* TC *is an exponentially* 1*-resilient protocol for* **tc**; *in particular, Phase I of* TC *is (exponentially) as* 1*-resilient as Phase I of* ID(**tc**), and *Phase II of* TC *is (exponentially) as* 1*-resilient as Phase II of* ID(**tc**).

Proof. The intuitive arguments given above are formalized as follows. An *S-defective* run occurs when a corrupt S undetectedly sends I and R tables that disagree on at least $k/2$ rows:

$$S\text{-defective} \Leftrightarrow |\{i|(\exists j)\,(cypher_{S\to I}) \neq b_I \cdot (key(i) \oplus j) + pad(i,j)\}| \geq \frac{k}{2}$$

and I does not broadcast CHEATING,

where $cypher_{S\to I}$ is the message sent by S to I. An *I-defective* run occurs when S is honest but R fails to detect an extensive attempt by I to cheat:

$$I\text{-defective} \Leftrightarrow |\{i|(\exists j)\,(cypher_{I\to R}) \neq b_I \cdot (key(i) \oplus j) + pad(i,j)\}| \geq \frac{k}{2}$$

and R does not broadcast CHEATING,

where $cypher_{I\to R}$ is the message sent by I to R. A *defective* run is one that is either *S-defective* or *I-defective*. The probability of a defective run is at most

$$\binom{3k/2}{k}\binom{2k}{k}^{-1} + 2^{-k/2},$$

which is $O(c^{-k})$ for some $c > 1$.

Thus, by Theorem 9 it suffices to show that the strong principle of independence holds for TC and ID(**tc**). Messages from nonfaulty players to faulty players must be considered.

The messages in round 1 from nonfaulty S to I are always uniformly random, as are those from S to R. If both I and R are corrupt, interface $\mathscr{I}$ must request that I and R send 1 to the trusted host in protocol ID(**tc**), thereby obtaining b, from which it directly computes the round 1 messages from S in TC. A nonfaulty I produces a uniformly random set of indices in round 2. A nonfaulty R sends strings received from S; if S is corrupt, interface $\mathscr{I}$ fills in the messages according to earlier specifications by $\mathscr{A}$, whereas if S is nonfaulty, the earlier message from S to R is produced simply by running S with the condition that b_I and *cypher* have the values specified already by the view of corrupt player I. (Interface $\mathscr{I}$ selects *key* at random and solves for *pad*.) In round 3, a nonfaulty I accepts if S and R are nonfaulty; otherwise, it is computable from previous corrupted messages whether nonfaulty I reports CHEATING or not.

The message from nonfaulty I to R in round 4.1 is determined by previous messages from S if S is corrupt. Otherwise, interface $\mathscr{I}$ corrupts R in protocol ID(**tc**)

(i.e., if it has not already done so) in order to obtain b. Using the value of b_R from round 1.3, $\mathcal{I}$ computes b_I, and using *key* and *pad* from round 1.3, $\mathcal{I}$ computes *cypher*, thereby determining the message from nonfaulty S to faulty R in round 4.1. If a nonfaulty R detects $k/2$ inconsistencies in round 5 ($\mathcal{I}$ calculates this from the adversarial view), then $\mathcal{I}$ causes I to send 1 in round 3 of the ideal protocol $\mathsf{ID}(\mathbf{tc})$, so that in the ideal protocol, R will also output CHEATING. Otherwise, I has behaved well enough in TC for R to accept the value S sent, and $\mathcal{I}$ causes a corrupt I in $\mathsf{ID}(\mathbf{tc})$ to send 0, allowing the trusted host to pass on the value of b.

By Theorem 9, then, $\mathrm{TC} \geq^{e} \mathsf{ID}(\mathbf{tc})$. □

5.1.2. *VSS.* The methods of [28] and [29] use time-encapsulated information to support VSS as follows. Combined with Shamir's method for secret sharing, TC allows a dealer to weakly share a secret: the dealer's secret is never compromised, and the dealer cannot change the secret once shared, but the dealer must remain present for the secret to be reconstructed. The dealer D uses TC with every pair (i, j), to give $\mathrm{PIECE}_i(s)$ to intermediate player i so that i can later report it to j.

To build a full VSS scheme from weak secret sharing, [28] and [29] use a second-level sharing of secrets. Each player i weakly shares $\mathrm{PIECE}_i(s)$ using subpieces $\mathrm{PIECE}_1(\mathrm{PIECE}_i(s)), \ldots, \mathrm{PIECE}_n(\mathrm{PIECE}_i(s))$. If a subdealer fails during the reconstruction of its piece $\mathrm{PIECE}_i(s)$, then that piece is omitted. The secretly shared pieces $\mathrm{PIECE}_1(s), \ldots, \mathrm{PIECE}_n(s)$ cannot be changed. With an additional cut-and-choose technique, the players can verify that the dealer D actually uses a proper polynomial to share s. If not, the dealer is disqualified, which constitutes a valid output that a dealer, even in the ideal case, can choose to make. Section 3.2 describes fault-recovery in general protocols; normally, if the result of a VSS scheme is that the dealer is disqualified, the dealer is written out of the overall protocol and any secret information it holds is reconstructed and revealed, so that all nonfaulty players can run an identical internal copy.

The details of the construction of a VSS protocol from a time-capsule protocol can be found in [29]. Although definitions of security are not provided in [29], the arguments given there essentially show that the strong principle of independence (as defined in Section 4.7) holds. By Theorem 9 of this paper, the arguments of [29] for the security of the VSS protocol apply *mutatis mutandis* to show the following:

Theorem 13 (After [29]). *For $t(n) < n/2$, there exists a t-resilient protocol* $\mathrm{VSS}(D, s)$ *for dealer D to verifiably secret share s. In particular, there exist protocols* SHARE *and* RECONSTRUCT *such that* (SHARE, RECONSTRUCT) *is an* $(\lfloor n/2 \rfloor)$*-threshold scheme.*

5.2. *Linear Combinations*

The principle behind adding secrets is simple (see [13]–[15]): if polynomials $p_{x_1}(u)$ and $p_{x_2}(u)$ represent secrets x_1 and x_2 (i.e., $p_{x_1}(0) = x_1$ and $p_{x_2}(0) = x_2$), then $p_{x_1}(u) + p_{x_2}(u)$ represents $x_1 + x_2$. Note that $(p_{x_1} + p_{x_2})(0) = x_1 + x_2$, the degree of $(p_{x_1} + p_{x_2})(u)$ is t, and if $p_{x_1}(u)$ or $p_{x_2}(u)$ is uniformly random then so is their sum. Hence it suffices to use $\mathrm{PIECE}_i(x_1 + x_2) = \mathrm{PIECE}_i(x_1) + \mathrm{PIECE}_i(x_2)$.

The robustness of a representation of a secret, however, is not based solely on the

piece each player holds but on the verification information from Vss. This includes the second-level sharing of each piece, as well as time-capsule verification information (*key*, *mask*, *pad*, *cypher*). As before, each subpiece of player i's new piece $\text{PIECE}_i(x_1 + x_2)$ is easily calculated as $\text{PIECE}_j(\text{PIECE}_i(x_1 + x_2)) = \text{PIECE}_j(\text{PIECE}_i(x_1)) + \text{PIECE}_j(\text{PIECE}_i(x_2))$.

It remains to show that each sender (namely each player i who has encapsulated various pieces $\text{PIECE}_j(\text{PIECE}_i(x_1))$ and $\text{PIECE}_j(\text{PIECE}_i(x_2))$ for other players j) can construct verification information for the sum of the encapsulated values. In general, say that sender S has encapsulated b_1 and b_2 using TC. The verification information is not compatible for adding b_1 and b_2. On the other hand, if the *keys* were identical, then the intermediary could simply add the masked (*cypher*) tables together and the recipient could add the pads together. Each would also add its b_I or b_R values together. With an informal notation representing the tables used in both cases:

$$\begin{aligned} cypher_{b_1} + cypher_{b_2} &= (mask_{b_1} + pad_{b_1}) + (mask_{b_2} + pad_{b_2}) \\ &= (mask_{b_1} + mask_{b_2}) + (pad_{b_1} + pad_{b_2}) \\ &= mask_{b_1+b_2} + pad_{b_1+b_2}. \end{aligned}$$

It is not hard to see that the resulting tables provide a proper verification structure for $(b_1 + b_2)$.

It is extremely improbable that the keys from each encapsulation will happen to be identical. The sender can, however, extend the tables further, using a key that is now identical in both cases. Two verifications of the sender's behavior are necessary. First, the extended tables must continue to represent b_1 and b_2. Second, intermediary I should, as usual, be assured that it can later convince R of the values. The second check is performed as in steps 2–4 of TC. At the same time, intermediary I can immediately deduce whether the rows returned are consistent with the b_I values it holds.

It is also easy to see that multiplication of encapsulated values and of pieces by a fixed constant is sufficient to produce a robust and private representation of the result of multiplying the encapsulated value or secret by that constant. For example, $\text{PIECE}_i(cx_1) = c\,\text{PIECE}_i(x_1)$ gives a piece derived from a polynomial $cp_{x_1}(u)$ representing x_1. For encapsulated values, no interaction or extension of structures is necessary.

Protocol LINEAR-COMBINE, described in Fig. 10, reflects the modifications to the verification structure. The only interactions in protocol LINEAR-COMBINE concern the extension of the verification structure and employ steps of TC; the proof of Theorem 12 demonstrates how an interface operates during these steps.

Theorem 14. *Protocol* LINEAR-COMBINE *computes* **LinComb** *t-resiliently.*

5.3. *Multiplication Resilient Against Passive Adversaries*

The idea behind secret addition fails when applied to multiplication. The value $\text{PIECE}_i(x_1) \cdot \text{PIECE}_i(x_2)$ is indeed a point on a polynomial $p_y(u) = p_{x_1}(u)p_{x_2}(u)$ with free term $p_y(0) = x_1x_2$, but the degree of $p_y(u)$ has grown to $2t$. Soon, there will not be enough pieces to determine the secret. In order to preserve the properties of VSS,

ADDPAIR (x_1, x_2, y)

1	$(1 \le D, i, j \le n)$	D, i, j:	$\text{TC}(D, i, j, \text{PIECE}_D(x_1))$, $\text{TC}(D, i, j, \text{PIECE}_D(x_2))$ with same new key^{Dij} (i checks this during TC) and with same old b_L, b_R
2.1	$(1 \le D \le n)$	D:	$\text{PIECE}_D(y) \leftarrow \text{PIECE}_D(x_1) + \text{PIECE}_D(x_2)$
2.2	$(1 \le D, i, j \le n)$	i:	$(mask + pad)^{D,i,j,y} \leftarrow (mask + pad)^{D,i,j,x_1} + (mask + pad)^{D,i,j,x_2}$
2.3	$(1 \le D, i, j \le n)$	i:	$(pad)^{D,j,i,y} \leftarrow (pad)^{D,j,i,x_1} + (pad)^{D,j,i,x_2}$

LINEAR-COMBINE $(\langle \Sigma, c_0, \ldots, c_W \rangle, x_1, \ldots, x_W, y)$

1	$(1 \le D, i, j \le n)$	D, i, j:	$\text{TC}(D, i, j, \text{PIECE}_i(x_1)), \ldots, \text{TC}(D, i, j, \text{PIECE}_i(x_W))$ with $key^{D,i,j,x_1} = key^{D,i,j,x_2} = \cdots = key^{D,i,j,x_W}$ (i checks this during TC) and with same old b_L, b_R
2.1	$(1 \le D \le n)$	D:	$\text{PIECE}_D(y) \leftarrow c_0 + \sum_{k=1}^{W} c_k \text{PIECE}_D(x_k)$
2.2	$(1 \le D, i, j \le n)$	i:	$(mask + pad)^{D,i,j,y} \leftarrow c_0 + \sum_{k=1}^{W} c_k (mask + pad)^{D,i,j,x_k}$
2.3	$(1 \le D, i, j \le n)$	i:	$(pad)^{D,j,i,y} \leftarrow c_0 + \sum_{k=1}^{W} c_k (pad)^{D,j,i,x_k}$

Fig. 10. Protocols to add or perform linear combinations (the function **LinComb**) of secret values. The ADDPAIR protocol illustrates the slightly more complicated LINEAR-COMBINE protocol. The superscripts (D, i, j, x) indicate the variables in the subprotocols with the given participants. Adding a constant to an array means to add the constant to each entry, and multiplying a constant by an array means to multiply each entry.

a new polynomial representing $x_1 x_2$ must have degree t. Without worrying about Byzantine errors, a simple protocol for degree reduction can be constructed from linear combinations (see [14] and [15]).

Any polynomial $q(u)$ of degree at most $(n-1)$ can be expressed as the weighted sum of fixed, LaGrange polynomials:

$$L_i(u) = \prod_{l=1,\ldots,n; l \neq i} \frac{(u-l)}{(i-l)},$$

$$q(u) = \sum_{i=1}^{n} L_i(u) q(i).$$

For any polynomial $q(u)$, define $\bar{q}(u) = q(u) \bmod u^{t+1}$. This operation truncates high-order terms. If player i knows and shares or has shared the value $q(i)$, then, for any j, $\bar{q}(j)$ is easily computed as a secret sum using protocol LINEAR-COMBINE:

$$\bar{q}(u) = \sum_{i=1}^{n} \bar{L}_i(u) q(i),$$

$$\bar{q}(j) = \sum_{i=1}^{n} \bar{L}_i(j) q(i).$$

UNIFSECRET (r)

1	$(1 \le i \le n)$	$r_i \leftarrow uniform(E)$
2	$(1 \le i \le n)$	Run VSS(i, r_i).
3		Run LINEAR-COMBINE: $r = \sum_{i=1}^{n} r_i$.

Fig. 11. Protocol to produce a uniformly random secret field element r.

TRUNCATE $(q(u))$

0			Each player i starts with secretly shared $q(i)$. Define $\alpha_{ij} = \bar{L}_i(j)$.
1			Run UNIFSECRET(R).
2	$(1 \le i \le n)$		Run Vss to extend PIECE$_i(R)$ to a fully shared secret, denoted $r(i)$.
3	$(1 \le j \le n)$		Run LINEAR-COMBINE: $s_j \leftarrow \sum_{i=1}^{n} \alpha_{ij} q(i) + \sum_{i=1}^{n} i \cdot \alpha_{ij} \cdot r(i)$.
4	$(1 \le i, j \le n)$	$i \to j$:	PIECE$_i(s_j)$, with verification information.
5	$(1 \le i \le n)$	i:	Use s_j as PIECE$_i(q(0))$.

Fig. 12. Protocol to produce pieces of a degree-t polynomial from pieces of a degree-$2t$ polynomial $q(u)$. At the start, it is assumed that each player i holds the secretly shared value $q(i)$.

For a particular j, the publicly known weights on secrets $q(1), \ldots, q(n)$ are $\bar{L}_1(j), \ldots, \bar{L}_n(j)$.

A simple way to "randomize" the coefficients of $q(u)$ without changing the free term is to add a polynomial of degree $t + 1$ and free term 0:

$$\{r(u) \leftarrow UPoly(t + 1, 0): \bar{q}(u) + \bar{r}(u)\} = UPoly(t, q(0)).$$

The TRUNCATE protocol described in Fig. 12 uses these linear combination and randomization methods to create a properly shared representation of $q(0)$. Letting $q(u) = p_{x_1}(u)p_{x_2}(u)$, the protocol to multiply secrets is simply to apply the truncation protocol to $q(u)$.

The values of PIECE$_i(x_1)$PIECE$_i(x_2)$ do not, however, provide a *robust* representation of the inputs to the truncation protocol: it is easy for a Byzantine adversary to break the protocol merely by changing one value of $q(i) =$ PIECE$_i(x_1)$PIECE(x_2). Each player must therefore supply PIECE$_i(x_1)$PIECE$_i(x_2)$ as a properly shared input and must prove that it supplies the correct input. Given that PIECE$_i(x_1)$ and PIECE$_i(x_2)$ are already weakly shared under Vss, if their sharing can be extended to a full-fledged VSS-sharing, it would then suffice for each player to prove that it has shared PIECE$_i(x_1)$PIECE$_i(x_2)$ as the input to protocol TRUNCATE. The chance of a defective run, i.e., a run in which an incorrect value of $q(i)$ is used without detection, must be made small. This task, called the ABC Problem, is the crucial step in achieving general multiparty protocols. A solution that uses VSS and linear combinations is given below.

5.4. *ABC Problem: Verifying Multiplication*

The key algorithmic result of this paper is a technique to solve the problem of verifying products of secrets:

The ABC Problem. Alice knows the values of secrets a and b. Alice must share a new secret c and prove to the other players that the secret value of c is ab, without revealing the values themselves.

In the ideal protocol ID(**abc**) for this problem, each player supplies the trusted host with its portion (namely, the pieces and verification information) of a robust and private representation of secrets a and b. Alice supplies the entire representation

PROVE-PRODUCT (a, b, c)

0			Let a and b verifiably secretly shared.
Phase I.			
1			Run Vss(Alice, c) if c not yet shared.
2	$(1 \le j \le 2k)$	Alice:	$r_j \leftarrow uniform(E)$
	$(1 \le j \le 2k)$	Alice:	$s_j \leftarrow uniform(E)$
	$(1 \le j \le 2k)$	Alice:	$d_j \leftarrow (a + r_j)(b + s_j)$
3	$(1 \le j \le 2k)$		Run Vss(Alice, r_j), Vss(Alice, s_j), Vss(Alice, d_j).
Phase II. Verify that $(\forall j)\ d_j = (a + r_j)(b + s_j)$.			
4	$(1 \le i \le 2k)$		Run UNIFSECRET(j_i).
5	$(1 \le i \le 2k)$		Run RECONSTRUCT(j_i).
6			$Y \leftarrow \{j_1, \ldots, j_k\}$, the first k distinct indices.
7	$(\forall j \in Y)$		Run LINEAR-COMBINE: $a_j \leftarrow a + r_j$.
8	$(\forall j \in Y)$		Run LINEAR-COMBINE: $b_j \leftarrow b + s_j$.
9	$(\forall j \in Y)$		Run RECONSTRUCT(a_j), RECONSTRUCT(b_j), RECONSTRUCT(d_j).
10			**if** $(\exists j \in Y)\ d_j \neq a_j b_j$ **then** disqualify Alice
Phase III. Verify that c matches ab.			
11	$(\forall j \notin Y)$		Run RECONSTRUCT(r_j), RECONSTRUCT(s_j).
12	$(\forall j \notin Y)$		Run LINEAR-COMBINE: $c_j \leftarrow c - d_j + as_j + br_j + r_j s_j$.
13	$(\forall j \notin Y)$		Run RECONSTRUCT(c_j).
14			**if** $(\exists j \notin Y)\ c_j \neq 0$ **then** disqualify Alice
15			**if** Alice not disqualified **then output** *accept*

Fig. 13. Protocol to prove that the product of two secrets is a third secret.

of c (i.e., all pieces); $c \neq ab$ indicates she quits, and $c = ab$ indicates that the host should give the pieces of c to the players.

Lemma 15 (ABC Lemma). *Let $t(n) < n/2$. If* (SHA, REC) *is a t-threshold scheme and* LINEAR-COMBINE *t-resiliently computes* **hide(LinComb)**, *then there is a t-resilient protocol* PROVE-PRODUCT *for* ID**(abc)**.

Proof. Figure 13 describes the protocol. In the first phase, Alice shares several triples (r, s, d) of secrets such that $D = (a + r)(b + s)$, which are used to check the behavior of Alice. In the second phase, the players select and reconstruct some of these values to confirm that every triple satisfies the equation. In the third phase, a simple linear combination uses the unrevealed triples of secrets to verify that $c = ab$. The protocol uses a subprotocol, UNIFSECRET, to generate uniformly random secretly shared values, which can later be reconstructed to generate a fair coin. (When the field E has characteristic 2, the low-order bit of the secret may be used as a uniformly random bit.) Given that the LINEAR-COMBINE protocol is $((n - 1)/2)$-resilient for **hide(LinComb)**, Theorem 10 shows that UNIFSECRET is as resilient as ID$(uniform(E))$, the ideal protocol for a function returning a uniformly random field element.

If Alice is nonfaulty, the strong principle of independence is satisfied. Every r and s is uniformly distributed, so the revealed sums $a_j = (a + r_j)$ and $b_j = (b + s_j)$ are also uniformly distributed, and hence the products $d_j = (a + r_j)(b + s_j)$ have a fixed distribution. Because UNIFSECRET resiliently computes $uniform(E)$, the generation of k random indices is resilient and the results are always uniformly distributed. The

value of every d_j matches $(a + r_j)(b + s_j)$. Furthermore, every secret c_j is 0, and no nonfaulty player sends an impeachment of Alice.

If Alice is faulty, it remains the case that the messages output by nonfaulty players are a fixed function of the results of the tests in phase II and phase III, which in turn depend only on the messages of Alice and a fixed, uniform distribution (the output of UNIFSECRET).

Since the strong principle of independence is satisfied, it remains to show that defective runs are scarce. A defective run is one in which Alice cheats (by sharing an incorrect c) but is not caught, or in which the sequence of $2k$ random field elements does not contain a subset of k distinct indices. Assume that Alice shares $c \neq ab$. Let X be the set of indices j for which $d_j = (a + r_j)(b + s_j)$, namely for which Alice has behaved. The set Y of indices chosen by the players must be a subset of X, or Alice would be disqualified. Indices $j \notin Y$ must satisfy $c_j = c - d_j + as_j + br_j + r_j s_j = 0$ or else Alice is caught. But $c \neq ab$ implies $X = \bar{Y}$, and the probability of this is at most $\binom{2k}{k}^{-1}$. Given that a sequence of $2k$ uniformly random field elements fails to produce a set of k distinct indinces with probability at most 2^{-k}, it is easy to see that

$$\textbf{PrDefect}(\text{PROVE-PRODUCT}) \leq \binom{2k}{k}^{-1} + 2^{-k} = O(c^{-k})$$

for some $c > 1$. An interface ensures identical outputs on nondefective runs by corrupting whomever $\mathscr{A}$ corrupts and by corrupting Alice in $\mathsf{ID}(\textbf{abc})$ if she is corrupt in PROVE-PRODUCT, in order to decide whether Alice must send a proper $c = ab$ to the trusted host or whether Alice cheats in PROVE-PRODUCT and must therefore send an improper value in $\mathsf{ID}(\textbf{abc})$ to cause the players to output CHEATING.

Theorem 9 applies; hence PROVE-PRODUCT $\geq^{e}$ $\mathsf{ID}(\textbf{abc})$. □

5.5. *Multiplication Resilient Against Byzantine Adversaries*

Given resilient protocols for polynomial truncation and to prove products of secrets, the multiplication protocol is straightforward. The protocol MULTIPLY is described in Fig. 14.

Theorem 16. *Protocol* MULTIPLY *t-resiliently computes* **hide**($\times$).

MULTIPLY-ONE (u, v, w)

1	$(1 \leq i \leq n)$	$q(i) \leftarrow \text{PIECE}_i(u)\text{PIECE}_i(v)$.
	$(1 \leq i \leq n)$	Run VSS$(i, q(i))$.
2	$(1 \leq i \leq n)$	Run PROVE-PRODUCT$(\text{PIECE}_i(u), \text{PIECE}_j(u), q(i))$.
3		Run TRUNCATE(q) to construct shared secret $w = q(0) = uv$.

MULTIPLY $(\langle \Pi, j, k \rangle, x_1, \ldots, x_w, y)$

1	Run MULTIPLY-ONE(x_j, x_k, y).

Fig. 14. Protocols to multiply secrets. Protocol MULTIPLY specifies indices of secrets to multiply, for use in circuit evaluation.

Proof. The proof that the UNIFSECRET protocol (Fig. 11) resiliently computes a uniformly random secret over field E follows directly from the resilience of Vss and of LINEAR-COMBINE, and using the fact that, since $t < n$, at least one player supplies a uniformly random addend, r_i. The resilience of TRUNCATE (Fig. 12) follows from the resilience of UNIFSECRET, LINEAR-COMBINE, and VSS, assuming the input values $q(i)$ lie on a polynomial of degree at most $2t$; the combined protocol TRUNCATE ∘ PROVE-PRODUCT ensures that defective runs have exponentially small probability. By Theorems 5 and 7, MULTIPLY is exponentially resilient. □

6. Zero-Knowledge Proofs

The notion of *resilience* captures zero-knowledge in a very concise manner. Consider not a two-party protocol but a *three*-party protocol for players P, V, and trusted host TH. Let L be a language and let χ_L: $\Sigma^* \to \{0, 1\}$ be its characteristic function, namely $\chi_L(x) = 1 \Leftrightarrow x \in L$. In the *ideal zero-knowledge proof system*, denoted $\mathsf{ID}_3(L)$, player P sends x to TH, who computes $\chi_L(x)$ and sends "$x \in L$" to V if $\chi_L(x) = 1$. The host otherwise sends "?" to indicate a failed proof.

The classical definition of a zero-knowledge proof system given by Goldwasser *et al.* [23] is then captured by a single sentence: A two-part protocol $\Pi = \langle P, V \rangle$ is a *zero-knowledge proof system* (ZKPS) for L iff it is as resilient as $\mathsf{ID}_3(L)$. When an adversary corrupts P, the resilience of the protocol ensures that a nonfaulty V never accepts a false statement "$x \in L$" (soundness), whereas if P remains nonfaulty, then a nonfaulty V always accepts "$x \in L$" (completeness). Furthermore, the trusted host passes on only "$x \in L$" to V, so the information gained by an adversary who corrupts V is limited to that knowledge. Perfect and statistical zero-knowledge are captured by perfect and statistical relative resilience. Computational zero-knowledge is captured by requiring that an interface $\mathcal{I}$ be a probabilistic polynomial-time machine, that the adversary $\mathcal{A}$ is also such a machine, and that the resulting ensembles are computationally indistinguishable.

Notice that a corrupt P does have *influence* over the output of V: it can cause V not to believe a proof or it can convince V of a true statement, but its influence is bounded by that permitted in the ideal case. The limits of the ideal case, along with the idea of relative resilience, capture all desired properties at once.

By designing the appropriate ideal protocol, other versions of zero-knowledge are equally easily defined. For example, a two-sided ZKPS is defined with respect to the ideal two-sided ZKPS, $\mathsf{ID}(\chi_L)$: P sends x to TH, who sends $\chi_L(x)$ to V (P also has the option to abstain by sending Λ, in which case TH sends Λ to V).

Ben-Or, Goldwasser, Kilian, and Wigderson introduced ZKPS in which many, physically seprate provers are used. In this paper the power of multiple parties is used to accomplish zero-knowledge proofs, but the parties are not required to be separate; instead, it is assumed that a majority remain nonfaulty.

Network Zero-Knowledge Proof Systems (NZKPS) are defined as concisely as ZKPS. In the idea case, $\mathsf{ID}(L)$, one party P sends x to trusted host TH, who sends on "$x \in L$" to V if $x \in L$, but otherwise sends "?". Variations are possible: many verifiers may be considered, or proofs systems on *secrets* may be considered, in

which x is a list of secretly shared values that are not revealed even though the the membership of x in L is revealed.

Most variations are covered by *Network Secret Zero-Knowledge Proof Systems* (NSZKPS). The ideal NSZKPS, $\mathsf{ID}(x_1, \ldots, x_N, y, \chi_L)$, contains a trusted host who broadcasts "yes" if $\chi_L(x_1, \ldots, x_N, y) = 1$ but otherwise broadcasts "?". Player P shares y and the players supply a robust (e.g., secretly shared) representation of $x_1, \ldots, x_n$. An NSZKPS for L is an n-party protocol Π such that $\Pi \geq \mathsf{ID}(x_1, \ldots, x_N, y, \chi_L)$.

Theorem 17. *For any $L \in NP$ and $t(n) < n/2$, there exists an exponentially t-resilient NSZKPS for L having round complexity $O(t)$ and message complexity and local complexity polynomial in n, m, k (with the possible exception of prover P, who may require an NP-computation to compute χ_L). For $t(n) < n/3$ there exists a perfectly t-resilient NSZKPS that uses a constant number of rounds.*

Proof. Let C_L be a circuit with nondeterministic inputs $\omega_1, \ldots, \omega_W$ that computes L, namely for which $x \in L$ iff there exists a setting of $\omega_1, \ldots, \omega_W$ such that $C_L(\omega_1, \ldots, \omega_W, x) = 1$. A list of a satisfying set of values is sufficient to prove $x \in L$. A simple NSZKPS would be for P to share the values of $\omega_1, \ldots, \omega_W$ and to run a multiparty protocol to evaluate C_L. The proof is accepted iff the results is 1.

This direct method requires many rounds of interaction, however. Interaction is usually a costly resource in distributed computing. An improved method is the following. Prover P shares not only $\omega, \ldots, \omega_W$ but the results of each gate $g(i, j)$ in C_L. Now, the network need not evaluate the gates; it need only verify that each output is correct with respect to its secret inputs. Linear combination gates are simple to check: the network runs the LINEAR-COMBINE protocol to compute $(x_{li} - \mathbf{LinComb}(c_0, \ldots, c_W, x_{(l-1)1}, \ldots, x_{(l-1)W}))$, and then reconstructs the result. If the result is nonzero, the proof fails. The PROVE-PRODUCT protocol is used to verify multiplicative gates, with P taking the part of Alice. It is not hard to show that protocols computing *secret* representations can be run in parallel. The number of rounds of a single subprotocol is on the order of t: each time a player faults detectably, the recovery algorithm must be run, but otherwise the number of rounds is constant. If $t(n) < n/3$, the protocols of [14] and [15] are used, giving a constant number of rounds regardless of Byzantine behavior. □

Theorem 18. *For any $L \in IP$ and $t(n) < n/2$, there exists an exponentially t-resilient NSZKPS for L. (For $t(n) < n/3$ there exists a perfectly t-resilient NSZKPS.)*

Proof. Without loss of generality [1], [24], there exists an Arthur–Merlin game for L, in which Arthur merely sends $p(n)$ random bits during each of $q(n)$ rounds, and afterward performs a polynomial-time computation F on the messages from Merlin. The NSZKPS for L is as follows. Repeat the following, $q(n)$ times. Run $p(n)$ copies of the UNIFSECRET protocol to generate $p(n)$ random bits, which are considered as Arthur's message to Merlin; player P then shares a message that Merlin would send to Arthur, given the sequences of messages and random bits generated thus far.

After $q(n)$ executions are completed, run EVAL to compute F on the set of messages shared by P and the $p(n)q(n)$ random bits. If the result of F is 1, then V accepts; otherwise, V rejects. Clearly, this process ideally computes the result of the interaction between Merlin and Arthur without revealing any of the messages. Because the NSZKPS protocol is a concatenation of polynomially many exponentially t-resilient subprotocols, it is itself exponentially t-resilient. □

7. Conclusion

This paper presents efficient and provably secure methods for a network of n parties to compute arbitrary finite functions at private arguments supplied by each party, revealing only the result. These methods tolerate $t < n/2$ faults, namely any faulty minority, the maximal achievable bound. The faulty parties may be chosen by a dynamic, resource-unbounded Byzantine adversary, who can rush messages before choosing new parties to corrupt. Each protocol allows an exponentially small chance of error, but it is easily proven that when $t \geq n/3$ no protocol can tolerate t Byzantine faults without error. No unproven assumptions are necessary. A complete network with private channels and a broadcast channel is assumed, however.

The protocols of this paper rely on a new technique to prove that the value of one secret is the product of the values of two others, a problem called the ABC Problem. A solution to the ABC Problem is presented that uses any technique for verifiable secret sharing that supports secret addition of secrets.

Ben-Or [29] has independently developed a solution to this problem, but the solution is restricted to logical bit-operations. The protocols in this paper permit arithmetic over large fields at the same cost, thereby achieving far fewer rounds of interaction for most natural problems. In distributed systems, the number of rounds of interaction is often the most significant resource. An interesting and practical open question is whether the number of rounds can be significantly reduced. Bar-Ilan and Beaver [2] give a method for reducing the number of rounds by a logarithmic factor, but it is not clear that even Verifiable Secret Sharing can be performed in a constant number of rounds when the number of corrupted parties may exceed a third of the network.

The methods used to solve the ABC Problem generalize to support efficient zero-knowledge proofs of any language in NP and, with additional machinery, any language in IP. The use of a network of communicating parties to maintain validity and privacy in the proof system, as opposed to physically separate provers, is novel.

The results of this paper are proven using a new and concise set of definitions for multiparty protocol security. Unlike previous definitions, which analyze separate properties of the desired solution and are in general too incohesive to be satisfying, the definitions used here are unified under the broad and powerful concept of *relative resilience*. Relative resilience provides a means to reduce one protocol to another in terms of security. It is the proper generalization of the basic idea of zero-knowledge to an interactive setting in which the *influence* of an adversary, not just the *information* it gains, must be considered.

Privacy, correctness, independence of inputs, and all desired properties are cap-

tured *a priori* by comparing a protocol to an ideal protocol in which an incorruptible host performs the computation. Because the ideal protocol is, by definition, the desired goal, any protocol which is equally resilient also captures the desired properties. Witness the difference between the original definition of zero-knowledge, which requires the description of several properties, and the equivalent definition requiring one line of text: $\langle P, V \rangle \geq \mathsf{ID}(L)$, i.e., the proof system must be as resilient as the ideal protocol guaranteed to give correct results.

Because protocol resilience satisfies convenient properties such as transitivity and invariance under protocol concatenation, it provides modular and understandable proofs of security. Using the notion of resilience, this paper also presents the first proofs that the common share/compute/reveal paradigm is valid. It also formalizes many folk theorems that are false without specific and easily overlooked conditions. A tremendous source of insecurity in practical settings arises precisely from instances where intuitions and unproven beliefs have turned out incorrect. These proofs give the precise conditions under which the protocols are secure, perhaps the most important factor in considering their value.

References

[1] L. Babai, S. Moran. Arthur–Merlin Games: A Randomized Proof System, and a Hierarchy of Complexity Classes. *J. Comput. System Sci.* **36** (1988), 254–276.

[2] J. Bar-Ilan, D. Beaver, Non-Cryptographic Fault-Tolerant Computing in a Constant Expected Number of Rounds of Interaction. *Proc. PODC*, ACM, New York, 1989, pp. 201–209.

[3] D. Beaver. Secure Multiparty Protocols Tolerating Half Faulty Processors. *Proceedings of Crypto 1989*, ACM, New York, 1989. Also appeared as Technical Report TR-19-88, Harvard University, September, 1988.

[4] D. Beaver. Perfect Privacy for Two-Party Protocols. *Proc. DIMACS Workshop on Distributed Computing and Cryptography*, Princeton, NJ, October, 1989, J. Feigenbaum, M. Merritt (eds.). Preliminary version in Technical Report TR-11-89, Harvard University.

[5] D. Beaver. Formal Definitions for Secure Distributed Protocols *Proc. DIMACS Workshop on Distributed Computing and Cryptography*, Princeton, NJ, October, 1989, J. Feigenbaum, M. Merritt (eds.).

[6] D. Beaver. Security, Fault Tolerance, and Communication Complexity in Distributed Systems. Ph.D. Thesis, Harvard University, 1990.

[7] D. Beaver, J. Feigenbaum. Hiding Instances in Multioracle Queries. *Proc.* 7th *STACS*, Lecture Notes in Computer Science, vol. 415, Springer-Verlag, Berlin, 1990, pp. 37–48. Also appeared as Hiding Information from Several Oracles, Technical Report TR-10-89, Harvard University, May 1, 1989.

[8] D. Beaver, J. Feigenbaum, J. Kilian, P. Rogaway. Cryptographic Applications of Locally Random Reductions. *Proc. Crypto 1990.* Also appeared as AT&T Bell Laboratories Technical Memorandum, November 15, 1989.

[9] D. Beaver, J. Feigenbaum, V. Shoup. Hiding Instances in Zero-Knowledge Proof Systems. *Proc. Crypto 1990.*

[10] D. Beaver, S. Goldwasser. Multiparty Computation with Faulty Marjority. *Proc.* 30th *FOCS*, IEEE, New York, 1989, pp. 468–473.

[11] D. Beaver, S. Haber, M. Yung. Protocols Secure Against Dynamic Adversaries. In preparation, 1990.

[12] D. Beaver, S. Micali, P. Rogaway. The Round Complexity of Secure Protocols. *Proc.* 22nd *STOC*, ACM, New York, 1990, pp. 503–513.

[13] J. Benaloh. Verifiable Secret Ballot Elections. Ph.D. Thesis, Yale University, 1987.

[14] M. Ben-Or, S. Goldwasser, A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. *Proc.* 20th *STOC*, ACM, New York, 1988, pp. 1–10.
[15] D. Chaum, C. Crépeau, I. Damgård. Multiparty Unconditionally Secure Protocols. *Proc.* 20th *STOC*, ACM, New York, 1988, pp. 11–19.
[16] B. Chor, S. Goldwasser, S. Micali, B. Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. *Proc.* 17th *STOC*, ACM, New York, 1985, pp. 383–395.
[17] B. Chor, E. Kushilevitz. A Zero–One Law for Boolean Privacy. *Proc.* 21st *STOC*, ACM, New York, 1989, pp. 62–72.
[18] Z. Galil, S. Haber, M. Yung. Cryptographic Computation: Secure Fault-Tolerant Protocols and the Public-Key Model. *Proc. Crypto 1987*, Springer-Verlag, Berlin, 1988, pp. 135–155.
[19] Z. Galil, S. Haber, M. Yung. Minimum-Knowledge Interactive Proofs for Decision Problems. *SIAM J. Comput.* **18**: 4 (1989), 711–739.
[20] O. Goldreich, S. Micali, A. Wigderson. Proofs that Yield Nothing but Their Validity and a Methodology of Cryptographic Protocol Design. *Proc.* 27th *FOCS*, IEEE, New York, 1986, pp. 174–187.
[21] O. Goldreich, S. Micali, A. Wigderson. How to Play Any Mental Game, or A Completeness Theorem for Protocols with Honest Majority. *Proc.* 19th *STOC*, ACM, New York, 1987, pp. 218–229.
[22] S. Goldwasser, L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. *Proc. Crypto 1990.*
[23] S. Goldwasser, S. Micali, C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.* **18**: 1 (1989), 186–208.
[24] S. Goldwasser, M. Sipser. Private Coins vs. Public Coins in Interactive Proof Systems. *Proc.* 18th *STOC*, ACM, New York, 1986, pp. 59–68.
[25] S. Haber, S. Micali. Personal communication, 1987.
[26] J. Kilian, S. Micali, P. Rogaway. The Notion of Secure Computation. Unpublished manuscript, 1990.
[27] E. Kushilevitz. Privacy and Communication Complexity. *Proc.* 30th *FOCS*, IEEE, New York, 1989, pp. 416–421.
[28] T. Rabin. Robust Sharing of Secrets When the Dealer is Honest or Cheating. Masters Thesis, Hebrew University, 1988.
[29] T. Rabin, M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. *Proc.* 21st *STOC*, ACM, New York, 1989, pp. 73–85.
[30] P. Rogaway. The Round Complexity of Secure Protocols. Ph.D. Thesis, Massachusetts Institute of Technology, 1990.
[31] A. Shamir. How To Share a Secret. *Comm. ACM* **22** (1979), 612–613.