Secure and Efficient Asynchronous Broadcast Protocols

Christian Cachin Klaus Kursawe Frank Petzold* Victor Shoup

IBM Research
Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{cca,kku,sho}@zurich.ibm.com

March 7, 2001

Abstract

Reliable broadcast protocols are a fundamental building block for implementing replication in fault-tolerant distributed systems. This paper addresses secure service replication in an asynchronous environment with a static set of servers, where a malicious adversary may corrupt up to a threshold of servers and controls the network. We develop a formal model using concepts from modern cryptography, present modular definitions for several broadcast problems, including reliable, atomic, and secure causal broadcast, and present protocols implementing them. Reliable broadcast is a basic primitive, also known as the Byzantine generals problem, providing agreement on a delivered message. Atomic broadcast imposes additionally a total order on all delivered messages. We present a randomized atomic broadcast protocol based on a new, efficient multi-valued asynchronous Byzantine agreement primitive with an external validity condition. Apparently, no such efficient asynchronous atomic broadcast protocol maintaining liveness and safety in the Byzantine model has appeared previously in the literature. Secure causal broadcast extends atomic broadcast by encryption to guarantee a causal order among the delivered messages. Threshold-cryptographic protocols for signatures, encryption, and coin-tossing also play an important role.

*Frank Petzold has since left IBM and can be reached at petzold@hepe.com.

Contents

1 Introduction

Broadcast protocols are a fundamental building block for fault-tolerant distributed systems. A group of servers can offer some service in a fault-tolerant way by using the state machine replication technique, which will mask the failure of any individual server or a fraction of them. In the model with Byzantine faults considered here, faulty servers may exhibit arbitrary behavior or even be controlled by an adversary.

In this paper, we present a modular approach for building robust broadcast protocols that provide reliability (all servers deliver the same messages), atomicity (a total order on the delivered messages), and secure causality (a notion that ensures no dishonest server sees a message before it is scheduled by the system). An important building block is a new protocol for multi-valued Byzantine agreement with "external validation." Our focus is on methods for distributing secure, trusted services on the Internet with the goal of increasing their availability and security. Cryptographic operations are exploited to a greater extent than previously for such protocols because we consider them to be relatively cheap, in particular compared to the message latency on the Internet.

We do not make any timing assumptions and work in a purely asynchronous model with a static set of servers and no probabilistic assumptions about message delays. Our protocols rely on a trusted dealer that is used once to set up the system, but they do not use any additional external constructs later (such as failure detectors or stability mechanisms). We view this as the standard cryptographic model for a distributed system with Byzantine faults. These choices maintain the safety of the service even if the network is temporarily disrupted. This model also avoids the problem of having to assume synchrony properties and to fix timeout values for a network that is controlled by an adversary; such choices are difficult to justify if safety and also security depend on them.

Despite the practical appeal of the asynchronous model, not much research has concentrated on developing efficient asynchronous protocols or implementing practical systems that need consensus or Byzantine agreement. Often, developers of practical systems avoid the approach because of the result of Fischer, Lynch, and Paterson [17], which shows that consensus is not reachable by protocols that use an a priori bounded number of rounds, even with crash failures only. But the implications of this result should not be overemphasized. In particular, there are randomized solutions that use only a *constant expected* number of rounds to reach agreement [35, 8, 4]. Moreover, by employing modern, efficient cryptographic techniques, this approach has recently been extended to a practical yet provably secure protocol for Byzantine agreement in the cryptographic model that withstands the maximal possible corruption [7].

The basic broadcast protocols (following Bracha and Toueg [6]) are *reliable broadcast*, which ensures that all servers deliver the same messages, and a variation of it that we call *consistent broadcast*, which only provides agreement among the actually delivered messages. Consistent broadcast is particularly useful in connection with a *verifiability* property for the delivered messages, which ensures that a party can transfer a "proof of delivery" to another party in a single piece of information. We describe message- and communication-efficient implementations of reliable and consistent broadcast based on cryptographic techniques, such as digital signatures and threshold signatures. Both of these broadcast primitives do not ensure agreement on messages from faulty senders, however, for which a Byzantine agreement protocol is needed.

The efficient randomized agreement protocols mentioned before work only for binary decisions (or for decisions on values from small sets). In order to build distributed secure applications, this is not sufficient. One also needs agreement on values from large sets, in particular for ordering multiple messages. We propose a new *multi-valued Byzantine agreement* protocol with an *external* validity condition and show how it can be used for implementing *atomic broadcast.* External validity ensures that the decision value is acceptable to the particular application that requests agreement; this corrects a drawback of earlier agreement protocols for multi-valued agreement, which could decide on illegal values. Both protocols use digital signatures and additional cryptographic techniques.

The multi-valued Byzantine agreement protocol invokes only a constant number of binary Byzantine agreement sub-protocols on average and achieves this by using a cryptographic common coin protocol in a novel way. It withstands the maximal possible corruption of up to one third of the parties and has expected quadratic message complexity (in the number of parties), which is essentially optimal.

Our atomic broadcast protocol guarantees that a message from an honest party cannot be delayed arbitrarily by an adversary as soon as a minimum number of honest parties are aware of that message. The protocol invokes one multi-valued Byzantine agreement per batch of payload messages that is delivered. An analogous reduction of atomic broadcast to consensus in the crash-fault model has been described by Chandra and Toueg [10], but it cannot be directly transferred to the Byzantine setting.

We also define and implement a variation of atomic broadcast called *secure causal atomic broadcast.* This is a robust atomic broadcast protocol that tolerates a Byzantine adversary and also provides secrecy for messages up to the moment at which they are guaranteed to be delivered. Thus, client requests to a trusted service using this broadcast remain confidential until they are answered by the service. This is crucial in our asynchronous environment for applying the state machine replication method to services that involve confidential data.

Secure causal atomic broadcast works by combining an atomic broadcast protocol with robust threshold decryption. The notion and a heuristic protocol were proposed by Reiter and Birman [39], who called it "secure atomic broadcast" and also introduced the term "input causality" for its properties. Recent progress in threshold cryptography allows us to present an efficient robust protocol together with a security proof in the appropriate formal models from cryptography.

In accordance with the comprehensive survey of fault-tolerant broadcasts by Hadzilacos and Toueg [21], we define and implement our protocols in a modular way, with reliable and consistent broadcasts and Byzantine agreement as primitives. This leads to the following layered architecture:

<table>
<tr><td colspan="2">Secure Causal Atomic Broadcast</td></tr>
<tr><td colspan="2">Atomic Broadcast</td></tr>
<tr><td colspan="2">Multi-valued Byzantine Agreement</td></tr>
<tr><td>Broadcast Primitives</td><td>Byzantine Agreement</td></tr>
</table>

Important for the presentation of our broadcast protocols is our formal model of a modular protocol architecture, where a number of potentially corrupted parties communicate over an insecure, asynchronous network; it uses complexity-theoretic concepts from modern cryptography. This makes it possible to easily integrate the formal notions for encryption, signatures, and other cryptographic tools with distributed protocols. The model allows for quantitative statements about the running time and the complexity of protocols; the essence of our definition is to bound the number of steps taken by participants on behalf of a protocol *independently* from network behavior. In view of the growing importance of cryptography for secure distributed protocols, a unified formal model for both is a contribution that may be of independent interest.

1.1 Related Work

The use of cryptographic methods for maintaining consistent state in a distributed system has a long history and originates with the seminal work of Pease, Shostak, and Lamport [33]; Dolev and Strong [14] derive lower bounds for protocols that use authentication. However, much of the early work on Byzantine agreement predates the development of robust and efficient cryptographic protocols and the adequate formal models, from which we benefit here.

Chandra and Toueg [10, p. 248] mention that Byzantine agreement and atomic broadcast are equivalent in asynchronous systems, but do not give any further details. In fact, we are not aware of any previous description of a protocol for asynchronous atomic broadcast with Byzantine faults in the literature.

A large body of research in distributed systems focuses on *view-based group communication systems* like Isis, Transis, or Horus for state machine replication [41] in the crash-fault model (see the overview in [34]). Such systems provide the abstraction of a process group, which may change over time. They guarantee certain synchrony properties among group members so that they all see the same messages; Vitenberg et al. [44] survey and compare various specifications found in the literature and implemented in practical systems.

Rampart [37, 38] is the only one of them that tolerates arbitrary (Byzantine) failures. It also uses cryptography for efficient reliable and atomic broadcasts [36], but solves a technically different problem than the one we address here: As Rampart builds on a membership protocol to agree dynamically on the group's composition, it cannot guarantee an honest majority within the group when facing an adversary that completely controls communication. Because the maintenance of safety is the primary application of our protocols for trusted services, such behavior cannot be tolerated and we have to use more expensive agreement methods.

Another class of protocols circumvent the impossibility of consensus in asynchronous environments by assuming a probabilistic behavior of the network links [6, 29, 31]. In particular, Moser and Melliar-Smith [31] present algorithms to obtain a total order from a partial order imposed by an underlying communication system. However, this model is not suitable for applications that need high security guarantees because such assumptions are rather difficult to justify in practice.

Castro and Liskov [9] describe a practical algorithm for state-machine replication that maintains safety despite Byzantine faults and its implementation for realizing a fault-tolerant distributed file system. Since their protocols are deterministic, however, this approach cannot ensure liveness at the same time—at least not without making certain timing assumptions.

SecureRing [22] and the very recent work of Doudou, Garbinato, and Guerraoui [15] are two examples of atomic broadcast protocols that rely on failure detectors in the Byzantine model. They encapsulate all time-dependent aspects in the abstract notion of a failure detector and permit clean, deterministic protocols. However, most implementations of failure detectors will use timeouts and actually suffer from some of the problems mentioned above. It also seems that Byzantine failure detectors are not yet well enough understood to allow for precise definitions.

In summary, we think the cryptographic model with randomized Byzantine agreement is both practically and theoretically attractive, although it seems to have been somewhat overlooked in the past. The fact that randomized agreement protocols may not terminate with non-zero probability does not matter because this probability is negligible. Moreover, if a protocol uses authentication, digital signatures, or *any* cryptography at all, and the practical protocols mentioned above do so, a negligible probability of failure cannot be ruled out anyway.

1.2 Organization of the Paper

The remainder of the paper starts with a description of our cryptographic system model in Section 2, where also the necessary cryptographic primitives are introduced. The paper continues with definitions and protocols for reliable broadcast and consistent broadcast in Section 3. Section 4 introduces the notion of validated Byzantine agreement and presents two protocols for multi-valued validated Byzantine agreement. The definition and a protocol for atomic broadcast are given in Section 5, and for secure causal atomic broadcast in Section 6.

2 Model

This section describes a formal model for our modular protocol architecture, where a number of parties communicate over an insecure, asynchronous network, and where an adversary may corrupt some of them.

Our model differs in two respects from other models traditionally used in distributed systems with Byzantine faults:

1. In order to use the proof techniques of complexity-based cryptography [18], our model is *computational*: all parties and the adversary are constrained to perform only feasible, i.e., polynomial-time, computations. This is necessary for using formal notions from cryptography in a meaningful way.

2. We make no assumptions about the network at all and leave it under complete control of the adversary. Our protocols work only to the extent that the adversary delivers messages faithfully. In short, *the network is the adversary.*

The differences become most apparent in the treatment of termination, for which we use more concrete conditions that together imply the traditional notion of "eventual" termination.

We define termination by bounding a *statistic* measuring the amount of work that honest, uncorrupted parties do on behalf of a protocol; in particular, we use the communication complexity of a protocol for this purpose. Since the specification of a protocol requires certain things to happen under the condition that all protocol messages have been delivered, bounding the length (and also the number) of protocol messages generated by uncorrupted parties ensures that the protocol has actually terminated under this condition. In cryptography one proves security with respect to all polynomial-time adversaries, and we adopt this model here as well. Our notion of an *efficient* (deterministic) protocol requires that the statistic is bounded by a fixed polynomial, which is independent of the adversary. As we rely on randomization (for Byzantine agreement as well as for other things), we also define a corresponding probabilistic bound for randomized protocols; from this a bound on the expected running time of a protocol can be derived. Both of our notions are closed under modular composition of protocols, which is not trivial for randomized protocols.

Among the many established formal models for asynchronous distributed protocols, the I/O automata model of Lynch and Tuttle [26, 28, 27] seems to be the most general one. It has also been extended to allow for modeling of randomized protocols. But even though authentication and digital signatures have been used before in secure distributed protocols, apparently no adequate formal model has integrated both approaches before [27, p. 115].

2.1 Basic System Model

The security parameter of our computational security model is denoted by k. A function $\epsilon(k)$ is called *negligible* if for all $c > 0$ there exists a k_0 such that $\epsilon(k) < \frac{1}{k^c}$ for all $k > k_0$. We will consider negligible functions also in other parameters than in k, but we assume that the parameter of a negligible function is k if not explicitly mentioned otherwise. In this sense, a "negligible quantity" is a negligible function in the security parameter k. As k is sometimes not mentioned in other contexts either, keep in mind that all system parameters are bounded by polynomials in k.

2.1.1 Parties and Protocols

Multi-Party Protocols. An n-party protocol consists of a collection of n parties, $P_1, \ldots, P_n$, which are probabilistic interactive Turing machines that run in polynomial time (in k). Such a machine has two dedicated interfaces for reading incoming messages and writing outgoing messages. There is also an initialization algorithm, which is run by an additional party called the dealer; on input k, n, and t, it generates the state information that is used to initialize each party. For simplicity, assume $n \leq k$.

After initialization, a party P_i may be activated repeatedly with some input message. It will carry out some computation, update its state, possibly generate some output messages, and wait for the next activation.

We leave it to the adversary to choose n and t, but a specific protocol might impose its own restrictions (e.g., $t < n/3$). We can assume that the dealer includes these values, as well as the index i, in the initial state of P_i.

Our model includes a public-key infrastructure for digital signatures, i.e., the dealer generates a key pair for a digital signature scheme $\mathcal{S}$ for each party, and includes in the initial state of each party its private key and the public keys of all parties. The dealer initializes a fixed number of threshold cryptosystems as required by the implemented protocols.

The dealer may also generate a public output for information associated with the n-party protocol; this information may be useful for clients of a replicated service that is implemented by the n-party protocol.

Executions and the Adversary. As our network is insecure and asynchronous, protocol execution is defined entirely via the adversary. The adversary is a polynomial-time interactive Turing machine that schedules and delivers all messages and corrupts some parties.

After the initial setup phase, the adversary repeatedly activates a party with some input message and waits for the party to generate some output message(s). The output is given to the adversary and perhaps indicates to whom these messages should be sent, and the adversary may choose to deliver these messages faithfully at some time. But in general, the adversary chooses to deliver any message it wants, or no message at all; we sometimes impose additional restrictions on the adversary's behavior, however.

The adversary also *corrupts* t parties. W.l.o.g. any adversary that corrupts fewer than t parties can be converted into one that corrupts exactly t parties. This simplification seems justified for distributed systems with Byzantine faults where one cannot rely on the actions of a single, potentially corrupted party; all our intended applications are be based on the behavior of (a majority of) the uncorrupted parties.

One distinguishes between *static* and *adaptive* corruptions in cryptography: in the *static* corruption model, the adversary must decide whom to corrupt independently of the execution of the system, whereas in the *adaptive* corruption model, the adversary can adaptively choose whom to corrupt as the attack is ongoing, based on information it has accumulated so far. We adopt a *static* adversary in this work for using the threshold coin-tossing scheme and the Byzantine agreement protocol of Cachin, Kursawe, and Shoup [7], the threshold cryptosystem of Shoup and Gennaro [43], and the threshold signature scheme of Shoup [42]. All of these assume static corruptions. However, we believe that the protocols described here generalize immediately to adaptive security, given such primitives with adaptive security.

The adversary receives the initial state of the corrupted parties as produced by the dealer. Otherwise, the corrupted parties are simply absorbed into the adversary: we do not regard them as system components. Uncorrupted parties are called *honest*.

Our formal model leaves control over the application interface for invoking broadcasts and starting agreement protocols up to the adversary. The protocol definitions merely state that *if* the adversary invokes the protocol in a certain way—in the same way an intended application would do—*then* the protocol should satisfy some specific conditions. This reflects that applications might be partially influenced by an adversary, which might cause some security problems if this is not allowed. For simplicity, this application program interface is mapped onto the single messaging interface, as described below.

Modular Protocol Architecture. We describe a modular protocol architecture, in which multiple broadcasts and transactions may execute in parallel. These protocol instances run concurrently and may also invoke other protocol instances on their behalf as sub-protocols. The dynamic relation between all concurrently running protocol instances is given by a directed acyclic graph in which every sub-protocol points to its parent. The "root" protocols with no parents represent instances directly invoked by a user application; in our formal model, they are invoked by the adversary. All other protocol instances are invoked as sub-protocols of some already running parent instance.

To identify protocol instances, we assume that each instance is associated with a unique *tag* ID. The value ID is an arbitrary bit string whose structure and meaning are determined by a particular protocol and application; in our formal model, the tag of the root instances is chosen by the adversary because the adversary invokes them. Sub-protocols are identified by hierarchical tags of the form $ID|ID'|\ldots$. The tag value $ID|ID'$ typically identifies a sub-protocol of the parent protocol instance ID and is determined by the parent. The adversary may not introduce a new tag on its own if this extends any previously introduced tag, i.e., the set of tags specified for the root instances must be prefix-free.

2.1.2 Communication

Messages. The protocols are described in terms of a single communication interface to which the adversary delivers messages. Each party runs an internal *scheduler* that delivers messages to the protocol instance associated with the corresponding ID. The message interface is used in two different ways, however: to send and to receive messages via the network and as a placeholder for local invocation of sub-protocols. Syntactically, invoking a sub-protocol appears as if it were a request of the adversary in our formal model, as mentioned before. Since our protocol specifications guarantee certain behavior when requests come from an arbitrary adversary, an application using a sub-protocol can benefit from this universality, as long as it meets the requirements in the respective specification. The detailed mechanism for composing protocols is part of the scheduler described below.

There are three different types of messages: input actions, output actions, and protocol (implementation) messages.

Input and *output* actions represent local events; they provide local input or carry local output to or from a protocol instance, which might be a sub-protocol of an already running instance. On the "protocol stack" of the layered architecture, input and output actions travel vertically: inputs "down" to sub-protocols and outputs "up" to higher-layer protocols.

All other messages are *protocol messages*, generated and processed by the protocol implementation; they are intended for the peer instances running at other parties on the same level of the stack (directed "horizontally"). Protocol messages are internal implementation messages and they are distinct from the messages or requests actually disseminated as payloads of the broadcast protocols; those messages are sometimes explicitly called *payload* messages.

An *input action* is a message of the form

$$(ID, \texttt{in}, action, \dots),$$

where *action* is specific to the protocol and followed by arbitrary data. Input actions represent local invocations of a protocol, either as a root protocol instance by the adversary or as a sub-protocol of an already running protocol instance. An input action is used to request a service from the protocol instance. There is a special input action *open*, represented by

$$(ID, \texttt{in}, \texttt{open}, type),$$

which must precede any other input action with tag ID. When P_i processes such a message with tag ID for the first time, it initializes the instance; *type* specifies the type of the protocol being initialized. We say that P_i has *opened* a "channel" with tag ID or *activated* a "transaction" with tag ID. (Although it is a crucial element, we usually assume that it occurs implicitly before the first regular input action.)

An *output action* is a message of the form

$$(ID, \texttt{out}, action, \dots),$$

where *action* is again dependent on the particular service. These messages typically contain an output from the protocol instance to the calling entity. There is a special output action *halt*, represented by

$$(ID, \texttt{out}, \texttt{halt}),$$

after which no further messages tagged with ID are processed by the party. When P_i generates such a message with tag ID, we say that P_i has *halted* instance ID.

We stress that in a real protocol implementation, input and output actions both do not involve any real network communication and will be mapped onto local events being generated or processed by the calling entity. But in the formal model at least some of them are generated and received by the adversary.

The third type of message generated by P_i are *protocol messages* of the form

$$(ID, type, \dots)$$

with $type \notin \{\texttt{in}, \texttt{out}\}$. The idea is that such messages are delivered by the network to other parties, where they are processed by the corresponding protocol instance.

For simplicity, we shall not include origin and destination addresses in the body of protocol messages, and assume that this information is implicitly available to the receiving party.

Furthermore, we assume that all protocol messages are *authenticated*, which restricts the adversary's behavior as follows: if P_i and P_j are honest and the adversary delivers a protocol message M to P_j indicating that it was sent by P_i, then M was generated by P_i at some prior point in time. It is reasonable to build authentication into our model because it can be implemented very cheaply using standard symmetric-key cryptographic techniques [30].

Internal Scheduling. When a party is activated by the adversary, the incoming message is appended to a local *message buffer* and the internal *scheduler* is invoked. It delivers the message to the protocol instance associated with the corresponding ID. If no protocol associated with ID is running yet, the scheduler buffers all arriving messages until a corresponding instance has been opened. If the protocol instance has already halted, the message is discarded.

The applicable messages in the buffer are delivered to the protocol instances as follows. For each input action *open* with a tag ID that has not been opened before, a new protocol instance with the specified ID is initialized and the scheduler remembers that it was started over the network (i.e., by the adversary).

Each opened protocol instance executes as a separate thread, but at any point in time, at most one of them is active. Upon activation of a party, all protocol instances are in *wait states*. An instance enters this state by executing a **wait for** operation, specifying a condition defined on the message buffer and other local state variables under which it processes a message. Waiting instances become *ready* as soon as their condition is satisfied. Then one of the ready instances is scheduled to execute (arbitrarily, if more than one are ready), subject to the following restriction: An instance ID is not scheduled if any of its *children* in the dynamic protocol tree are also ready. In this way, instance ID is scheduled only *after* any other ready instance whose tag contains ID as a proper prefix.

When a protocol instance is scheduled, it processes the message, potentially generating some messages, until it enters the wait state again by issuing a **wait for** operation, or until it performs an explicit **halt** operation. The scheduler translates **halt** into the output action *halt* for tag ID and removes the instance ID (further messages tagged with ID are ignored).

The scheduler treats messages generated by an instance ID as follows. *Protocol messages* are simply written to the outgoing communication interface. For each input action *open*, however, a new protocol instance with the specified child ID is initialized, as if the message came from the network. The scheduler remembers the ID of the parent instance; all subsequent *input actions* from the parent addressed to the child are not written out to the network, but included directly in the buffer. Each *output action* of a sub-protocol instance ID is mapped directly into a corresponding internal message for its parent; *output actions* of a root protocol instance are written to the outgoing communication interface. These steps allow local activation of sub-protocols and local processing of their output to be described in terms of the single message interface.

The scheduler continues to deliver messages to waiting protocol instances in an arbitrary order, until the buffer contains no more applicable messages. When no more instance are ready, control is returned to the adversary. Some messages may remain in the buffer until the next activation because no protocol was waiting for them. Correctness and security of a protocol should not depend on the particular implementation of the scheduler, as long as it obeys these rules.

Our protocol descriptions are mostly written in reactive style, consisting simply of message handlers for which a global **wait for** operation is issued implicitly. Upon receiving an applicable message, the handler will execute some instructions, update its state, and may also perform a

wait for operation which will block until the appropriate messages have arrived. If an instance *ID* **waits for** messages tagged with *its own ID*, it is simply a shorthand notation for the corresponding message handlers. But if an instance *ID* **waits for** output from a child instance (that has previously been opened), the scheduler delivers the output actions of the child to the parent, as mentioned before. We make the assumption that an instance **waiting for** output from an uninitialized instance triggers implicitly a corresponding *open* action, which initializes the instance.

2.1.3 Quantitative Aspects

Defining Termination. In the model with computationally bounded participants considered here, we cannot apply the notion of "eventual" termination traditionally used in distributed computing, which allows for infinite protocol runs and would make formal models of cryptographic methods with computationally bounded adversaries meaningless. Instead, we define termination of a protocol instance only to the extent that the adversary faithfully delivers messages among the honest parties (analogous to [7]). In order to bound the adversary's running time, we quantify the amount of work done by honest parties on behalf of a protocol. We measure the *efficiency* of a protocol for this purpose. Combined with a liveness condition (such as "validity"), restricting the amount of work implies eventual termination in the conventional sense. For example, this will rule out trivial protocols that never terminate but always cause some work to be done without making progress.

Formally, our efficiency condition is based on a *protocol statistic* X measuring the work done *by honest parties* in a multi-party protocol execution, such as "useful" computation steps or the number of generated message bits. A protocol statistic is a family of real-valued, non-negative random variables $\{X_A(k)\}$, parameterized by adversary A and security parameter k, where each $X_A(k)$ is a discrete random variable induced by the coin flips of the dealer, the honest parties, and adversary A for security parameter k. We call X a *bounded protocol statistic* if for all adversaries A, there exists a polynomial p_A such that for all $k \geq 0$, it holds $X_A(k) \leq p_A(k)$, i.e., the statistic is polynomial in the security parameter, but depending on the adversary.

The key to defining efficiency lies in "uniformly" bounding a protocol statistic, *independent* of the adversary—such a bound should only depend on the particular protocol implementation. As we consider deterministic and randomized protocols (which may not always terminate after a polynomial number of steps), we introduce two corresponding notions for such uniformly bounded statistics.

Definition 1 (Uniformly Bounded Statistics). Let X be a bounded protocol statistic. We say that

1. X is *uniformly bounded (by T)* if there exists a fixed polynomial $T(k)$ such that for all adversaries A, there exists a negligible function $\epsilon_A(k)$ such that for all $k \geq 0$,

$$\Pr[X_A(k) > T(k)] \ \leq \ \epsilon_A(k);$$

2. X is *probabilistically uniformly bounded (by T)* if there exists a fixed polynomial $T(k)$ and a fixed negligible function $\delta(l)$ such that for all adversaries A, there exists a negligible function $\epsilon_A(k)$ such that for all $l \geq 0$ and $k \geq 0$,

$$\Pr[X(k) > lT(k)] \ \leq \ \delta(l) + \epsilon_A(k).$$

A probabilistically uniformly bounded statistic is allowed to exceed the uniform bound with non-negligible probability in the security parameter, but this probability must again be negligible, *independent* of the adversary. If X probabilistically uniformly bounded by T, then its expected value is bounded by T times a constant that is independent of the adversary, as shown next.

Lemma 1. *Suppose X is a statistic of a multi-party protocol that is probabilistically uniformly bounded by T. Then there exists a constant c such that for all adversaries A, the expected value of $X_A(k)$ is bounded by $cT(k) + \epsilon'_A(k)$, where ϵ'_A is a negligible function.*

Proof. Recall that a bounded protocol statistic is bounded by some polynomial $q_A(k)$ in the security parameter, depending on the adversary A; thus the random variable $X_A(k)$ exceeds $q_A(k)$ with probability zero.

Set $X'_A(k) = X_A(k)/T(k)$; it follows $X'_A(k) \leq q'_A(k)$ for some polynomial $q'_A(k)$. Because $X_A(k)$ is probabilistically uniformly bounded, we know that there exist negligible functions $\delta(l)$ and $\epsilon_A(k)$ such that $\Pr[X'_A(k) > l] \leq \delta(l) + \epsilon_A(k)$. Together with $\mathrm{E}[Y] \leq \sum_{l \geq 0} \Pr[Y > l]$ for any non-negative discrete random variable Y, it follows

$$\mathrm{E}[X'_A(k)] \;\leq\; \sum_{l \geq 0} \Pr[X'_A(k) > l] \;=\; \sum_{l=0}^{q'_A(k)} \Pr[X'_A(k) > l] \;\leq\; \sum_{l=0}^{q'_A(k)} \big(\delta(l) + \epsilon_A(k)\big).$$

Now fix δ to a function whose sum converges to a constant, say $\delta(l) = l^{-2}$. We have

$$\sum_{l=0}^{q'_A(k)} \big(\delta(l) + \epsilon_A(k)\big) \;\leq\; \sum_{l=0}^{q'_A(k)} l^{-2} + q'_A(k)\epsilon_A(k) \;\leq\; c_0 + q'_A(k)\epsilon_A(k)$$

for a constant c_0 that is independent of the adversary. Because ϵ_A is negligible and by the linearity of expectation, this implies that $\mathrm{E}[X_A(k)] = c_0 T(k) + \epsilon'_A(k)$ for some negligible ϵ'_A. □

A key property of these notions is that they lend themselves to the composition of protocols by way of the following lemma, whose proof is tedious but straightforward.

Lemma 2. *Fix any polynomial $F(x_1, \ldots, x_f)$, independent of adversary. If $X_1, \ldots, X_f$ are [probabilistically] uniformly bounded statistics, then $F(X_1, \ldots, X_f)$ is also a [probabilistically] uniformly bounded statistic.*

Communication and Message Complexity. An appropriate statistic in the above sense is the *communication complexity* of a multi-protocol; it is used by our model to define termination. Formally, the communication complexity is equal to the *bit length* of all *associated* protocol messages that *honest parties generate*. Which protocol messages are associated to a particular instance *ID* will vary according to the protocol type and will be noted explicitly when defining a protocol. Typically, this includes all messages with the tag *ID* or any tag starting with $ID|\ldots$; through the second form, also messages generated by sub-protocols on behalf of the calling protocol can be associated to an instance *ID*. Our protocol architecture ensures that all messages generated by honest parties are associated to some protocol.

Restricting the communication complexity to messages *generated* by *honest* parties seems the best one can say about a protocol in a Byzantine environment; the adversary can always deliver "junk" protocol messages to honest parties, which require some work to be read. Network bandwidth is an apparent resource that communication protocols consume, thus, measuring it seems adequate.

Alternatively, one could bound the bit length of all distinct messages delivered to one honest party that were generated by another honest party. But this is bounded by the communication complexity in the sense above.

As it is, there is no a priori restriction on the size of a payload message in our formal model. However, the communication complexity of a broadcast protocol depends on the length of such a message. For simplicity, we will therefore assume that there exists a fixed polynomial upper bound on the length of all payload messages that are contained in any input or output action message of any honest party. From this, and from the description of a particular protocol implementation, one can derive an upper bound on the maximal length of any protocol message.

Another appropriate statistic for a certain class of protocols (like Byzantine agreement, as used in [7]) is the *message complexity*, defined as the total *number* of all *associated* protocol messages that *honest parties generate*.

If the communication complexity (or also the message complexity) is uniformly bounded, the adversary *could* quickly make all honest parties terminate the protocol instance, but it is not *forced* to do so.

Modular Protocol Composition. Using the message complexity (or communication complexity) as a statistic has the advantage that it is closed under the *modular composition* of protocols as follows. According to our architecture, a higher-level protocol may invoke a sub-protocol to carry out a certain task; this appears as a one or more input actions generated by the higher-level protocol, which will start the sub-protocol(s) as described above. Suppose for the moment that sub-protocols are implemented by a distributed oracle available to every party, which provides the service of the sub-protocols in an ideal and instantaneous way. We call such a protocol an *oracle protocol*. A party invokes the protocol oracle by generating a suitable input action message, so that this counts as one towards message complexity.

Consider two multi-party oracle protocols A and B with respective message complexities X_A and X_B that are both [probabilistically] uniformly bounded. Suppose that the oracle protocol A uses an oracle for the task provided by B. Since B is implemented by the oracle, X_A counts every invocation of B by any honest party as one unit.

If we replace every oracle call on behalf of A to B by actually invoking B according to our general system model, we obtain a composed protocol AB with message complexity X_{AB}. This counts all messages that protocol A generates directly and those generated by the instances of B started on behalf of A. But because X_A and X_B are [probabilistically] uniformly bounded, there exist the appropriate polynomial bounds on the message complexities of A and B and also on the number of activations of protocol B (because message complexity bounds also the number of sub-protocol invocations). Thus, by Lemma 2, X_{AB} is also [probabilistically] uniformly bounded.

In other words, if we compose two, or any constant number of protocols with [probabilistically] uniformly bounded message complexities (some of them being oracle protocols), we obtain another protocol with [probabilistically] uniformly bounded message complexity. This extends trivially to communication complexity and, in fact, to any statistic in which invoking a sub-protocol is counted as one cost unit.

Lemma 3. *[Probabilistically] uniformly bounded communication complexity is closed under the modular composition of protocols.*

This is an important property of our notion of termination for randomized protocols and justifies the way in which we have defined it. If one would merely consider the *expected value* of a statistic for a randomized protocol, one could not draw such conclusions. For example, combining a protocol from which we only know that its expected number of rounds is constant with another one having the same property would not guarantee that the total expected number of rounds is also constant.

2.2 Byzantine Agreement

We give the definition of *Byzantine agreement* (or *consensus* in the crash-fault model) here as it is needed for building atomic broadcast protocols. It can be used to provide agreement on independent *transactions*.

The Byzantine agreement protocol is activated when the adversary delivers a message to P_i of the form

$$(ID, \texttt{in}, \texttt{propose}, v),$$

where $v \in \{0, 1\}$. When this occurs, we say P_i *proposes* v for transaction ID.

A party terminates the Byzantine agreement protocol (for transaction ID) by generating an output message of the form

$$(ID, \texttt{out}, \texttt{decide}, v).$$

In this case, we say P_i *decides* v for transaction ID.

Let any message with tag ID or $ID|\ldots$ that is generated by an honest party be *associated* to the agreement protocol for ID.

Definition 2 (Byzantine agreement). A protocol solves *Byzantine agreement* if it satisfies the following conditions except with negligible probability:

Validity: If all honest parties that are activated on a given ID propose v, then any honest party that terminates for ID decides v.

Agreement: If an honest party decides v for ID, then any honest party that terminates decides v for ID.

Liveness: If all honest parties have been activated on ID and all associated messages have been delivered, then all honest parties have decided for ID.

Efficiency: For every ID, the communication complexity for ID is probabilistically uniformly bounded.

This is the usual definition of validity in the literature. In Section 4 we introduce the weaker notion of *external validity* that is useful for certain applications. For instance, if initial values come with validating data (e.g., a digital signature) that establishes their validity in a particular context, we will require that an honest party may only decide on a value for which it has the accompanying validating data. Thus, even if all honest parties start with 0, they may still decide on 1 if they obtain the corresponding validating data for 1 during the agreement protocol.

2.3 Cryptographic Primitives

Apart from ordinary digital signature schemes, we use robust, non-interactive threshold signatures, threshold public-key encryption schemes, and a threshold coin-tossing protocol.

We need a *collision-free hash function* $H : \{0,1\}^* \to \{0,1\}^{k'}$ with the property that the adversary cannot generate two distinct strings x and x' such that $H(x) = H(x')$, except with negligible probability.

Another useful primitive is a cryptographically strong *pseudorandom generator* [19], denoted by $G : \{0,1\}^{k''} \to \{0,1\}^*$, that stretches a k''-bit seed by an arbitrary polynomial factor. G is a deterministic algorithm with input a random k''-bit seed such that its output is computationally indistinguishable from a uniform random string of the same length. In other words, for every efficient statistical test running in time polynomial in k, the probability that it can distinguish the output of G with a random seed from truly random bits is negligible.

Many efficient cryptographic schemes, and in particular all the threshold-cryptography protocols needed below, can be analyzed only in the so-called *random-oracle model* [1]. This refers to an idealized world where a hash function has been replaced by a truly random oracle, available to all participants. Although such proofs provide only a heuristic notion of security, the model allows to design truly practical protocols that admit a security analysis, which yields very strong evidence for their security.

2.3.1 Digital Signatures

A digital signature scheme [20] consists of a *key generation* algorithm, a *signing* algorithm, and a *verification* algorithm. The key generation algorithm takes as input a security parameter, and outputs a public key/private key pair. The signing algorithm takes as input that private key and a message m, and produces a signature σ. The verification algorithm takes the public key, a message m, and a putative signature σ, and outputs a bit that indicates whether it accepts or rejects the signature. A signature is considered *valid* if and only if the verification algorithm accepts. All signatures produced by the signing algorithm must be valid.

The basic security property is *unforgeability.* The attack scenario is as follows. An adversary is given the public key, and then requests the signatures on a number of messages, where the messages themselves may depend on previously obtained signatures. If at the end of the attack, the adversary can output a message m and a valid signature σ on m, such that m was not one of the messages whose signature it requested, then the adversary has successfully *forged* a signature. Security means that it is computationally infeasible for an adversary to forge a signature.

2.3.2 Non-Interactive Threshold Signatures

An important tool for our broadcast protocols are non-interactive threshold signatures. More precisely, we need *dual-threshold* variations as introduced by Shoup [42] and Cachin, Kursawe, and Shoup [7]. The basic idea of a dual-threshold signature scheme is that there are n parties, t of which may be corrupted. The parties hold *shares* of the secret key of a signature scheme, and may generate *shares of signatures* on individual messages. The only requirement is that κ signature shares are necessary and sufficient to construct a signature, where $t < \kappa \leq n - t$. (The standard notion of threshold schemes considers only $\kappa = t + 1$.)

More precisely, a non-interactive (n, κ, t)-dual-threshold signature scheme consists of the following parts:

- A *key generation algorithm* with input parameters k, n, κ, and t. It outputs the public key of the scheme, a private key share for each party, and a local verification key for each party.
- A *signing algorithm* with inputs a message, the public key and a private key share. It outputs a signature share on the submitted message.
- A *share verification algorithm* with inputs a message, a signature share on that message from a party P_i, along with the global public key and the local verification key of P_i. It determines if the signature share is valid.
- A *share combining algorithm* that takes as input a message and κ valid signature shares on the message, along with the public key and the verification keys, and (hopefully) outputs a valid signature on the message.
- A *signature verification algorithm* that takes as input a message and a signature (generated by the share-combining algorithm), along with the public key, and determines if the signature is valid.

The interaction takes place in the basic system model introduced above. During initialization, the dealer runs the key generation algorithm and gives each party the public key, all local verification keys, and its private key share. The adversary may submit signing requests to the honest parties for messages of its choice. Upon receiving such a request, a party computes a *signature share* for the given message using its private key share. Given κ valid signature shares from distinct parties on the same message, they may be combined into a signature on the message.

The two basic security requirements are *robustness* and *non-forgeability.* Robustness means that it is computationally infeasible for an adversary to produce κ valid signature shares such that the output of the share combining algorithm is not a valid signature. Non-forgeability means that it is computationally infeasible for the adversary to output a valid signature on a message that was submitted as a signing request to *less* than $\kappa - t$ honest parties.

A practical scheme that satisfies these definitions in the random-oracle model was proposed by Shoup [42] and is based on RSA [40]. Each signature share has essentially the size of an RSA signature and the final signature is a standard RSA signature. Our definition of a threshold signature scheme would also admit the trivial implementation of just using a set of κ ordinary signatures.

The dual-threshold scheme is used in some of our protocols, where a threshold signature with $\kappa > t + 1$ provides evidence for the fact that $\kappa - t$ honest parties have executed some steps in the protocol. A single-threshold scheme would not work here because although our system corruption model is static, the adversary may adaptively decide from which honest parties to request additional signature shares by scheduling messages accordingly.

2.3.3 Non-Interactive Threshold Cryptosystems

We use the definition of non-interactive threshold cryptosystems with security against adaptive chosen-ciphertext attacks put forward by Shoup and Gennaro [43]. (For ordinary public-key cryptosystems, security against adaptive chosen-ciphertext attacks is equivalent to non-malleability [13].)

A $(n, t + 1)$-threshold cryptosystem is given by the following algorithms:

- A *key generation algorithm*, taking as input k, n, and t. Outputs are the public key and a private decryption key for each party.

- An *encryption algorithm* with inputs the public key, a cleartext message $m \in \{0,1\}^*$. The algorithm outputs a ciphertext c and a label $\ell \in \{0,1\}^*$.

- A *decryption algorithm* with inputs the public key, an index $i \in \{1,\ldots,n\}$, the private key of P_i, a ciphertext c, and a label ℓ. It outputs a *decryption share* or a special symbol $\perp$ if the inputs are invalid.

- A *combination algorithm* that takes as inputs the public key, a ciphertext c, a label ℓ and a list D of decryption shares, of which some may be invalid. If D contains at least $t+1$ valid decryption shares, the algorithm outputs the cleartext m. Otherwise it returns a special symbol $\perp$.

The interaction takes place in the basic system model according to Section 2.1. During the initialization phase, the dealer runs the key generation algorithm and gives each party the global public key and its private key share.

Any party may run the encryption algorithm with the public key and a cleartext message to produce a ciphertext.

For decryption, a party sends the ciphertext together with the label to each party P_i, who returns a decryption share. Upon receiving enough decryption shares, the decryptor can combine them in order to obtain the cleartext.

The algorithms ensure that if a ciphertext c of a cleartext m was produced correctly by the encryption algorithm, then the recovery algorithm yields m with all but negligible probability, even if at most t decryption shares were not produced by the decryption algorithm with inputs as specified above. This property is called *robustness*.

To define security against adaptive chosen ciphertext attacks, consider the following game, played by the adversary in our basic system model with t statically corrupted parties; the keys generated by the dealer and given to the corrupted parties are seen by the adversary.

A1. The adversary interacts with the uncorrupted parties in an arbitrary fashion, feeding them ciphertext/label pairs and obtaining decryption shares.

A2. The adversary chooses two cleartexts, m_0 and m_1, and gives them to an "encryption oracle." The oracle chooses a bit b at random, encrypts m_b, and returns the resulting ciphertext c and label ℓ to the adversary.

A3. The adversary continues to interact with the uncorrupted parties, feeding them ciphertext/label pairs (c', ℓ') and receiving decryption shares, with the restriction that $(c', \ell') \neq (c, \ell)$.

A4. The adversary outputs a bit $\hat{b}$.

The threshold cryptosystem is called *secure against adaptive chosen ciphertext attack* if for any polynomial-time bounded adversary the probability that $b = \hat{b}$ exceeds $1/2$ only by a negligible quantity.

A practical threshold cryptosystem according to the above definition has been presented by Shoup and Gennaro [43]. Its security is based on the computational Diffie-Hellman problem [12], and it works in the random-oracle model; a variation of it is based on the decisional Diffie-Hellman problem.

2.3.4 Threshold Coin-Tossing

We also need an $(n, t+1)$-threshold coin-tossing scheme. The basic idea is the same as for the other threshold primitives, but here the parties hold *shares* of a pseudorandom function F. It maps a bit string N, the name of a coin, to its value $F(N) \in \{0,1\}^{k''}$. We use a generalized coin that produces k'' random bits simultaneously; such a coin is also called a distributed pseudo-random function [32]. The parties may generate *shares of a coin value* $F(N)$ and $t+1$ shares of the same coin are both necessary and sufficient to construct the value of that coin. The generation and verification of coin shares are also non-interactive and we work in the basic system model of Section 2.1.

During initialization the dealer generates a global verification key, a local verification key for each party, and a secret key share for each party. The initial state information for each party consists of its secret key share and all verification keys. The secret keys implicitly define a function F mapping names to k''-bit strings.

After the initialization phase, the adversary submits *reveal requests* to the honest parties for coins of his choice. Upon receiving such a request, a party outputs a *coin share* for the given coin computed from its secret key.

The coin-tossing scheme also specifies two algorithms:

- The *share verification* algorithm takes as input the name of a coin, a share of this coin from a party P_i, along with the global verification key and the verification key of P_i, and determines if the coin share is valid.

- The *share combining* algorithm takes as input a the name N of a coin and $t+1$ valid shares of N, along with (perhaps) the verification keys, and (hopefully) outputs $F(N)$.

The security requirements are *robustness* and *pseudorandomness.* Robustness means that it is computationally infeasible for an adversary to produce a name N and κ valid shares of coin N such that the output of the share combining algorithm is not $F(N)$. To define pseudorandomness, consider the following game, played in the basic system model.

D1. The adversary interacts with the uncorrupted parties in an arbitrary fashion, obtaining shares for arbitrary coins.

D2. The adversary chooses a coin N for which it has not yet requested a coin share, and gives it to an "F-oracle." The oracle chooses a bit b at random, and returns $F(N)$ if $b=0$ and a uniformly random k''-bit string otherwise.

D3. The adversary continues to interact with the uncorrupted parties and may obtain shares for arbitrary coins, except for N.

D4. The adversary outputs a bit $\hat{b}$.

The threshold coin-tossing scheme is *pseudorandom* if for any polynomial-time bounded adversary the probability that $b = \hat{b}$ exceeds $1/2$ only by a negligible quantity.

An efficient threshold coin-tossing scheme in the random-oracle model has been presented by Cachin, Kursawe, and Shoup [7]. Although their implementation produces single-bit outputs, it can be trivially modified to generate k''-bit strings, just by using a k''-bit hash function to compute the final value. Its security is based on the computational Diffie-Hellman problem in the random-oracle model. A related scheme for a distributed pseudo-random function, with security based on the decisional Diffie-Hellman problem, has also been proposed by Naor, Pinkas, and Reingold [32].

3 Broadcast Primitives

In this section, we introduce two broadcast primitives, *reliable broadcast* and *consistent broadcast*, and present communication-efficient protocols for both. In terms of our definitions, reliable broadcast (the Byzantine generals problem) appears as an extension of consistent broadcast; but we introduce reliable broadcast first because it is a well-known primitive. We also introduce the notion of a *verifiable* broadcast.

3.1 Reliable Broadcast

Reliable broadcast provides a way for a party to send a message to all other parties. It requires that all honest parties deliver the same set of messages and that this set includes all messages broadcast by honest parties, without guaranteeing anything about the order in which messages are delivered. In the context of arbitrary faults, reliable broadcast is also known as the *Byzantine generals problem* [25].

3.1.1 Definition

Broadcasts are parameterized by a tag ID, which can also be thought of as identifying a broadcast "channel." Since many parties can potentially broadcast several payload messages with the same ID, we augment the tag in a reliable broadcast by the identity of the sender, j, and by a sequence number s. Then, we restrict the adversary to submit a request for reliable broadcast tagged with $ID.j.s$ to P_i only if $i = j$ and at most once for every sequence number. These requirements are easily satisfied in practice by maintaining a message counter. Instances of reliable broadcast are always identified by $ID.j.s$ so that the simple tag ID alone represents a "virtual channel" for reliable broadcast; its implementation uses one independent protocol instance per payload message.

A reliable broadcast protocol is activated when the adversary delivers a message to P_j of the form

$$(ID.j.s, \texttt{in}, \texttt{r-broadcast}, m),$$

with $m \in \{0,1\}^*$ and $s \in \mathbb{N}$. When this occurs, we say P_j *reliably broadcasts* m *tagged with* $ID.j.s$, or simply P_j *r-broadcasts* m. Note that only P_j is activated like this. The other parties are activated when they perform an explicit *open* action for instance $ID.j.s$ in their role as receivers; according to our convention, this occurs for instance when they **wait for** an output tagged with $ID.j.s$.

A party terminates a reliable broadcast of m tagged with $ID.j.s$ by generating an output message of the form

$$(ID.j.s, \texttt{out}, \texttt{r-deliver}, m).$$

In this case, we say P_i *reliably delivers* m *tagged with* $ID.j.s$ (or *r-delivers* for brevity).

We say that all protocol messages which are generated by honest parties have tags with prefix $ID.j.s$ are *associated* to the broadcast of m by P_j with sequence number s. Recall that this defines also the messages contributing to the communication complexity of the protocol instance $ID.j.s$.

Definition 3 (Reliable Broadcast). A protocol for *reliable broadcast* satisfies the following conditions except with negligible probability:

Validity: If an honest party has *r-broadcast* m tagged with $ID.j.s$, then all honest parties *r-deliver* m tagged with $ID.j.s$, provided all honest parties have been activated on $ID.j.s$ and the adversary delivers all associated messages.

Consistency: If some honest party *r-delivers* m tagged with $ID.j.s$ and another honest party *r-delivers* m' tagged with $ID.j.s$, then $m = m'$.

Totality: If some honest party *r-delivers* a message tagged with $ID.j.s$, then all honest parties *r-deliver* some message tagged with $ID.j.s$, provided all honest parties have been activated on $ID.j.s$ and the adversary delivers all associated messages.

Integrity: For all ID, senders j, and sequence numbers s, every honest party *r-delivers* at most one message m tagged with $ID.j.s$. Moreover, if all parties follow the protocol, then m was previously *r-broadcast* by P_j with sequence number s.

Efficiency: For any ID, sender j, and sequence number s, the communication complexity of instance $ID.j.s$ is uniformly bounded.

Some remarks on the above definition. Recall the implicit quantification over all polynomial-time adversaries.

1. Validity ensures the liveness of a protocol, and rules out trivial protocols that do not generate any messages. One could use an equivalent, but simpler definition here, requiring that only the sender (and not all honest parties) *r-deliver* the message; but then one would have to modify this again to the present form for defining consistent broadcast below.

2. The agreement condition found in traditional definitions is split into consistency and totality. The reason for separating them is not only that they are distinct properties, but also that a reliable broadcast without a totality guarantee is a useful notion, as shown later.

3. The provision that the "adversary delivers all associated messages" is our quantitative counterpart to the traditional "eventual" delivery assumption. It can be ensured for an arbitrary adversary as follows. Suppose the adversary halts and there are yet undelivered protocol messages among honest parties (these can be inferred from a transcript of the adversary's interactions). Then using a "benign" scheduler delivering all the undelivered messages and the newly generated ones, the protocol is run until no more undelivered protocol messages exist, whereby termination in polynomial time is guaranteed by efficiency and validity.

4. Integrity may seem weak, since our model assumes authenticated links and we could hope to get the guarantee in the second clause also with t actually corrupted parties. Indeed, most reliable broadcast protocols implicitly also authenticate the sender of a message. It is possible to define the corresponding notion of an *authenticated* reliable broadcast by replacing the integrity condition above by the following.

   Authenticity: For all ID, senders j, and sequence numbers s, every honest party *r-delivers* at most one message m tagged with $ID.j.s$. Moreover, if P_j is honest, then m was previously *r-broadcast* by P_j with sequence number s.

However, we will not use authenticity in the standard definitions below because only some of our protocols provide authenticity. In particular, the protocols for reliable and for consistent broadcast provide authenticity, but not the atomic broadcast protocol.

We should note that an actual *implementation* of reliable broadcast is not needed by any of our protocols below. However, we build on the *definition* of reliable broadcast for defining other forms of broadcast. Nevertheless, we give a protocol for reliable broadcast in the next section—for completeness and to illustrate the system model and our definitions.

3.1.2 A Protocol for Reliable Broadcast

Protocol RBC **for party** P_i **and tag** $ID.j.s$

INITIALIZATION:

$\bar{m} \leftarrow \bot; \bar{d} \leftarrow \bot$

$e_d \leftarrow 0; r_d \leftarrow 0 \qquad (d \in \{0,1\}^{k'})$

UPON RECEIVING MESSAGE $(ID.j.s, \texttt{in}, \texttt{r-broadcast}, m)$:

send $(ID.j.s, \texttt{r-send}, m)$ to all parties

UPON RECEIVING MESSAGE $(ID.j.s, \texttt{r-send}, m)$ FROM P_l:

if $j = l$ **and** $\bar{m} = \bot$ **then**

$\bar{m} \leftarrow m$

send $(ID.j.s, \texttt{r-echo}, H(m))$ to all parties

UPON RECEIVING MESSAGE $(ID.j.s, \texttt{r-echo}, d)$ FROM P_l FOR THE FIRST TIME:

$e_d \leftarrow e_d + 1$

if $e_d = n - t$ **and** $r_d \leq t$ **then**

send $(ID.j.s, \texttt{r-ready}, d)$ to all parties

UPON RECEIVING MESSAGE $(ID.j.s, \texttt{r-ready}, d)$ FROM P_l FOR THE FIRST TIME:

$r_d \leftarrow r_d + 1$

if $r_d = t + 1$ **and** $e_d < n - t$ **then**

send $(ID.j.s, \texttt{r-ready}, d)$ to all parties

else if $r_d = 2t + 1$ **then**

$\bar{d} \leftarrow d$

if $H(\bar{m}) \neq d$ **then**

send $(ID.j.s, \texttt{r-request})$ to $P_1, \ldots, P_{2t+1}$

wait for a message $(ID.j.s, \texttt{r-answer}, m)$ such that $H(m) = \bar{d}$

$\bar{m} \leftarrow m$

output $(ID.j.s, \texttt{out}, \texttt{r-deliver}, \bar{m})$

UPON RECEIVING MESSAGE $(ID.j.s, \texttt{r-request})$ FROM P_l FOR THE FIRST TIME:

if $\bar{m} \neq \bot$ **then**

send $(ID.j.s, \texttt{r-answer}, \bar{m})$ to P_l

Figure 1: Protocol RBC for authenticated reliable broadcast (or the Byzantine generals problem) adopted from Bracha [5].

A message-efficient reliable broadcast protocol, denoted RBC, is given in Figure 1; it results from a small modification of Bracha's reliable broadcast protocol [5] to reduce the communication complexity.

Protocol RBC uses the hash of a payload message as a short, but unique representation for the potentially much longer message. The idea is that the payload is sent only once by the sender to all parties (similar to [36]). When a party is ready to deliver a payload message but does not yet know it, it asks an arbitrary subset of $2t+1$ parties for its contents and at least one of them will answer with the correct value.

In the description of the protocol, recall the global **wait for** condition for any message with a matching tag. Let $\perp$ denote a special value that cannot be broadcast. To implement the condition that a particular message from a party is processed only the first time it is received, one has to maintain the corresponding flags and counters, indexed by the contents of the message.

Theorem 4. *Assuming H is a collision-free hash function, Protocol* RBC *provides authenticated reliable broadcast for $n > 3t$.*

Proof. Validity is clear for honest senders by inspection of the protocol because all parties receive the initial `r-send` message and also $2t+1$ `r-ready` messages from honest parties, provided all associated messages are delivered. It may not hold for faulty senders, though.

For *consistency*, suppose an honest party P_i has *r-delivered* m and another honest party $P_{i'}$ has *r-delivered* $m' \neq m$ with tag $ID.j.s$. Then P_i must have received `r-ready` messages containing $d = H(m)$ from at least $t+1$ honest parties; the same holds for $P_{i'}$ with $d' = H(m')$. If $d = d'$, the adversary has created a collision in H. We assume no such collisions occur in the rest of the proof.

An honest party generates an `r-ready` message for d only if it has received $n-t$ `r-echo` messages containing d or $t+1$ `r-ready` messages already containing d. Thus, at least one honest party has sent an `r-ready` message containing d upon receiving $n-t$ `r-echo` messages; at most t of them are from corrupted parties. Similarly, some honest party must have received $n-t$ `r-echo` messages containing d'. Thus, there are at least $2(n-t) \geq n+t+1$ `r-echo` messages with tag $ID.j.s$ and at least $n-t+1$ among them from honest parties. But no honest party generates more than one such message by the protocol.

To establish *totality*, note that if some honest P_i delivers $\bar{m}$, then it has received the message $(ID.j.s, \texttt{r-ready}, \bar{d})$ from $2t+1$ different parties. Therefore, at least $t+1$ honest parties have sent `r-ready` with $ID.j.s$ and $\bar{d} = H(\bar{m})$, which will be received by all honest parties (assuming the adversary delivers all messages). Thus, all honest parties will send the corresponding `r-ready` message and any other party P_l will receive $2t+1$ of them. If P_l already knows m' with $H(m') = \bar{d}$, it outputs that.

Otherwise, P_l will send an `r-request` to $2t+1$ parties and wait for an `r-answer` satisfying $H(m') = \bar{d}$. Observe that there is at least one honest party who has sent an `r-ready` message containing $\bar{d}$ upon receiving $n-t$ corresponding `r-echo` messages. Thus, there are at least $n-2t$ honest parties who sent `r-echo` and know some m' such that $H(m') = \bar{d}$. Sending the `r-request` to $2t+1$ parties ensures that at least one of them receives and answers it, provided all messages are delivered.

For *integrity*, the uniqueness of the *r-delivered* message is clear from the protocol. If the sender P_j of message with sequence number s is honest, then at most t parties will send `r-echo` messages for tag $ID.j.s$ with $m' \neq m$. Thus, no uncorrupted party generates an `r-ready` message with d different from $H(m)$ and no uncorrupted party outputs m'. Actually, the protocol also satisfies *authenticity* because honest parties process `r-send` messages only from the sender indicated by the `r-echo` message.

It is easy to see that the protocol satisfies *efficiency* for any sender. □

Note that collecting $n - t$ `r-echo` messages is needed for totality (because `r-request` messages are sent to only $2t+1$ parties), but for consistency alone, this could be relaxed to $\lceil \frac{n+t+1}{2} \rceil$ `r-echo` messages.

The message complexity of Protocol RBC is $O(n^2)$. If messages are delivered faithfully by a "benign" scheduler and no faults occur, then its communication complexity is only $O(n^2 k' + n|m|)$ for broadcasting a single message m, where k' is the length of a hash value. However, the adversary can delay the `r-send` messages for some parties and increase the communication complexity. Since there are at most t honest parties who issue an `r-request` by the argument above to establish totality, m is transmitted $O(t^2)$ times and the overall communication complexity is $O(n^2 k' + n|m| + t^2|m|)$, or $O(n^2|m|)$ with maximal resilience.

Contrast this with the standard form of Bracha's broadcast that requires bit complexity $\Omega(n^2|m|)$, even in executions without faults. Under optimal circumstances, Protocol RBC needs to transmit m only once per party in the system.

3.2 Verifiable Broadcast

A party P_i that has delivered a payload message using reliable broadcast may want to inform another party P_j about this. Such information might be useful to P_j if it has not yet delivered the message, but can exploit this knowledge somehow, in particular since P_j is guaranteed to deliver the same message later by the agreement property. In a standard reliable broadcast, such as the protocol from the previous section, however, this knowledge cannot be transferred in a verifiable way.

We formalize this property of a broadcast protocol here because it is useful in our application below, and call it *verifiability.* Informally, it means this: when P_j claims that it is not yet in a state to deliver a particular payload message m, then P_i can reply with a single protocol message and when P_j processes this, it will deliver m immediately and terminate the corresponding broadcast.

Definition 4 (Verifiability). A broadcast protocol is called *verifiable* if the following holds, except with negligible probability: When an honest party has delivered m tagged with ID, then it can produce a single protocol message M that it may send to other parties such that any other honest party will deliver m tagged with ID upon receiving M (provided the other party has not already delivered m).

We call M the message that *completes* the verifiable broadcast. This notion implies that there is a predicate V_{ID} that the receiving party can apply to an arbitrary bit string for checking if it constitutes a message that completes a verifiable broadcast tagged with ID.

Protocol RBC could be made verifiable by adding a digital signature to the *r-ready* messages (this idea goes back to Pease, Shostak, and Lamport [33]). But verifiability is more useful in connection with weaker protocols than reliable broadcast; for example, in the consistent broadcast introduced next.

3.3 Consistent Broadcast

The totality property of reliable broadcast is rather expensive to satisfy; it is the main reason why most protocols for reliable broadcast need on the order of n^2 messages. For some applications, however, totality is not necessary and can be ensured by other means, as long as consistency and integrity are satisfied. We call the resulting notion *consistent broadcast* and discuss it in this section.

Several protocols for consistent broadcast have been proposed by Reiter et al. [36, 29]. To ensure agreement (i.e., totality) for delivered messages, these protocols are complemented by an external stability mechanism from which parties learn about the existence of messages they have not yet delivered. No such general mechanism is assumed here, but the parties may learn that also from an application.

3.3.1 Definition

The same restrictions on the adversary apply as for reliable broadcast. A consistent broadcast protocol is activated when the adversary delivers a message to P_j of the form

$$(ID.j.s, \texttt{in}, \texttt{c-broadcast}, m),$$

with $m \in \{0,1\}^*$ and $s \in \mathbb{N}$. When this occurs, we say P_j *consistently broadcasts* m *tagged with* $ID.j.s$.

A party terminates a consistent broadcast of m tagged with $ID.j.s$ by generating an output message of the form

$$(ID.j.s, \texttt{out}, \texttt{c-deliver}, m).$$

In this case, we say P_i *consistently delivers* m *tagged with* $ID.j.s$. To distinguish consistent broadcast from other forms of broadcast, we will sometimes use the terms *c-broadcast* and *c-deliver*.

All protocol messages generated by honest parties and tagged with $ID.j.s$ are associated to the broadcast of m by P_j with sequence number s.

Definition 5 (Consistent Broadcast). A protocol for *consistent broadcast* is a protocol for reliable broadcast that does not necessarily satisfy *totality*.

In other words, consistent broadcast makes no provisions that two parties do deliver the payload message, but maintains consistency among the actually delivered messages with the same senders and sequence numbers.

The notion of an authenticated consistent broadcast can be defined similarly to authenticated reliable broadcast, replacing the integrity condition by authenticity.

3.3.2 A Protocol for Verifiable Consistent Broadcast

Protocol VCBC implements verifiable consistent broadcast and is described in Figure 2. It uses a non-interactive $(n, \lceil \frac{n+t+1}{2} \rceil, t)$-dual-threshold signature scheme $\mathcal{S}_1$ with verifiable shares according to Section 2.3.2. Recall that all messages are authenticated according to our basic system model.

The protocol is based on the "echo broadcast" of Reiter [36], but uses a threshold signature to decrease the communication complexity. The idea behind it is that the sender broadcasts the message to all parties and hopes for $\lceil \frac{n+t+1}{2} \rceil$ parties to sign it as "witnesses" to guarantee consistency. The signature shares are then collected by the sender and combined to a threshold signature on the message; it then sends the signature all parties. After receiving the message together with a valid signature, a party delivers it immediately.

Protocol VCBC **for party** P_i **and tag** $ID.j.s$

Initialization:

$\bar{m} \leftarrow \bot; \bar{\mu} \leftarrow \bot$
$W_d \leftarrow \emptyset; r_d \leftarrow 0 \qquad (d \in \{0,1\}^{k'})$

Upon receiving message $(ID.j.s, \texttt{in}, \texttt{c-broadcast}, m)$:

send $(ID.j.s, \texttt{c-send}, m)$ to all parties

Upon receiving message $(ID.j.s, \texttt{c-send}, m)$ from P_l:

if $j = l$ **and** $\bar{m} = \bot$ **then**
$\bar{m} \leftarrow m$
compute an $\mathcal{S}_1$-signature share ν on $(ID.j.s, \texttt{c-ready}, H(m))$
send $(ID.j.s, \texttt{c-ready}, H(m), \nu)$ to P_j

Upon receiving message $(ID.j.s, \texttt{c-ready}, d, \nu_l)$ from P_l for the first time:

if $i = j$ **and** ν_l is a valid $\mathcal{S}_1$-signature share **then**
$W_d \leftarrow W_d \cup \{\nu_l\}$
$r_d \leftarrow r_d + 1$
if $r_d = \lceil \frac{n+t+1}{2} \rceil$ **then**
combine the shares in W_d to an $\mathcal{S}_1$-threshold signature μ
send $(ID.j.s, \texttt{c-final}, d, \mu)$ to all parties

Upon receiving message $(ID.j.s, \texttt{c-final}, d, \mu)$:

if $H(\bar{m}) = d$ **and** $\bar{\mu} = \bot$ **and** μ is a valid $\mathcal{S}_1$-signature **then**
$\bar{\mu} \leftarrow \mu$
output $(ID.j.s, \texttt{out}, \texttt{c-deliver}, \bar{m})$

Implementation of verifiability property

Upon receiving message $(ID.j.s, \texttt{c-request})$ from P_l:

if $\bar{\mu} \neq \bot$ **then**
send $(ID.j.s, \texttt{c-answer}, \bar{m}, \bar{\mu})$ to P_l

Upon receiving message $(ID.j.s, \texttt{c-answer}, m, \mu)$ from P_l:

if $\bar{\mu} = \bot$ **and** μ is a valid $\mathcal{S}_1$-signature on $(ID.j.s, \texttt{c-ready}, H(m))$ **then**
$\bar{\mu} \leftarrow \mu$
$\bar{m} \leftarrow m$
output $(ID.j.s, \texttt{out}, \texttt{c-deliver}, \bar{m})$

Figure 2: Protocol VCBC for verifiable and authenticated consistent broadcast.

Because a party may forward the message and the signature to other parties, the protocol is also verifiable according to Definition 4. The corresponding interface is implemented by the `c-request` and `c-answer` messages, which are not otherwise used by the protocol.

The consistency property of the protocol is based on the following lemma.

Lemma 5. *For all senders j, sequence numbers s, and strings ID, it is infeasible for the adversary in Protocol* VCBC *to create valid $\mathcal{S}_1$-signatures on the strings $(ID.j.s, \texttt{c-ready}, m)$ and $(ID.j.s, \texttt{c-ready}, m')$ with $m \neq m'$.*

Proof. Suppose not. Then, assuming $\mathcal{S}_1$ is secure, there are at least $\lceil \frac{n+t+1}{2} \rceil - t$ signature shares from distinct honest parties on a message containing $ID.j.s$ and m and at least as many from honest parties on the message containing $ID.j.s$ and m'. In total, there are $n+t+1-2t = n-t+1$ or more shares generated by honest parties containing $ID.j.s$. Since there are only $n-t$ honest parties, at least one honest party has signed two different messages with the same sender j and sequence number s, which is impossible according to the protocol. □

Theorem 6. *Assuming $\mathcal{S}_1$ is a secure $(n, \lceil \frac{n+t+1}{2} \rceil, t)$-dual-threshold signature scheme, Protocol* VCBC *provides verifiable and authenticated consistent broadcast for $n > 3t$.*

Proof. *Validity* for an honest sender is obvious from the construction of the protocol since all honest parties generate a signature share on m as soon as they receive an `c-send` message containing m. Since at least $\lceil \frac{n+t+1}{2} \rceil$ honest parties return them to the sender, it can combine them to a valid signature and *c-deliver* the message.

The *consistency* property follows directly from Lemma 5 because an honest party *c-delivers* a payload message only after verifying the corresponding threshold signature.

Integrity follows directly from Lemma 5 together with the logic of the protocol, where $\bar{\mu} \neq \perp$ is used to represent the state in which $\bar{m}$ has already been *c-delivered.* The protocol provides also *authenticity* because honest parties process `c-send` messages only from the sender indicated by the message.

Finally, *efficiency* is straightforward to verify and *verifiability* is ensured by the `c-answer` protocol message, which is generated upon receiving a suitable `c-request`. □

The message complexity of Protocol VCBC is $O(n)$ and its bit complexity is $O(n(|m| + K))$, assuming the length of a threshold signature and a signature share is at most K bits.

4 Validated Byzantine Agreement

The standard notion of Byzantine agreement implements a binary decision and can guarantee a particular outcome only if *all* honest parties propose the same value. We introduce in this section a weaker validity condition, called *external validity*, which relaxes the standard validity condition and generalizes to decisions on a value from an arbitrarily large set. It requires that the decided value satisfies a global predicate that is determined by the particular application and known to all parties. Each party adds some validation data to the proposed value, which serves as the proof for its validity. Typically, this consists of a digital signature that can be verified by all parties. The agreement protocol then returns to the caller not only the decision value, but also the corresponding validation data—the caller might need this information if it did not know it before. The standard validity condition is the special case of a trivially true predicate.

Validated Byzantine agreement generalizes the primitive of *agreement on a core set* [2, 3], which is used in the information-theoretic model for a similar purpose. Validated Byzantine agreement also generalizes the notion of *interactive consistency* [16] to the Byzantine model, which requires agreement on a vector of n values, one from each party.

Another related problem is *set agreement* [11], in which the agreement condition is relaxed so that the output of each party is contained in a small, global set. Although there exists a considerable literature on this problem, it cannot be used for our applications because it gives only an approximation of agreement.

4.1 Definition

Suppose there is a global polynomial-time computable predicate Q_{ID} known to all parties, which is determined by an external application. Each party may propose a value v together with a proof π that should satisfy Q_{ID}. The agreement domain is not restricted to binary values.

A validated Byzantine agreement protocol is activated by a message of the form

$$(ID, \texttt{in}, \texttt{v-propose}, v, \pi),$$

where $v \in \{0,1\}^*$ and $\pi \in \{0,1\}^*$. When this occurs, we say P_i *proposes* v *validated by* π for transaction ID. We assume the adversary activates all honest parties on a given ID at most once and, w.l.o.g., honest parties propose values with proofs that satisfy Q_{ID}.

A party terminates a validated Byzantine agreement protocol by generating a message of the form

$$(ID, \texttt{out}, \texttt{v-decide}, v, \pi).$$

In this case, we say P_i *decides* v *validated by* π for transaction ID.

We say that any protocol message with tag ID that was generated by an honest party is *associated* to the validated Byzantine agreement protocol for ID. An agreement protocol may also invoke sub-protocols for low-level broadcasts or for Byzantine agreement; in this case, all messages associated to those protocols that are started *on behalf* of the validated agreement protocol are associated to ID as well (such messages have tags with prefix $ID|\ldots$).

Definition 6 (Validated Byzantine Agreement). A protocol solves *validated Byzantine agreement* with predicate Q_{ID} if it satisfies the following conditions except with negligible probability:

External Validity: Any honest party that terminates for ID decides v validated by π such that $Q_{ID}(v, \pi)$ holds.

Agreement: If some honest party decides v for ID, then any honest party that terminates decides v for ID.

Liveness: If all honest parties have been activated on ID and all associated messages have been delivered, then all honest parties have decided for ID.

Integrity: If all parties follow the protocol, and if some party decides v validated by π for ID, then some party proposed v validated by π for ID.

Efficiency: For every ID, the communication complexity for ID is probabilistically uniformly bounded.

In other words, honest parties may propose all different values and the decision value may have been proposed by a corrupted party, as long as honest parties can verify the corresponding validation during the protocol. Note that agreement, liveness, and efficiency are the same as in the definition of ordinary, binary Byzantine agreement. Integrity is needed to rule out some trivial protocols in cases where a trivial predicate is used.

Another variation of the validity condition is that an application may prefer one decision value over others. Such an agreement protocol may be *biased* and *always* output the preferred value in cases where other values would have been valid as well.

For binary validated agreement, we will need a protocol that is biased towards 1 below. Its purpose is to detect whether there is a validation for 1, so it suffices to guarantee termination with output 1 if $t+1$ honest parties know the corresponding information at the outset. A *binary validated Byzantine agreement protocol biased towards 1* is a protocol for validated Byzantine agreement on values in $\{0, 1\}$ such that the following condition holds:

Biased External Validity: If at least $t + 1$ honest parties propose 1, then any honest party that terminates for ID decides 1.

We describe two related protocols for multi-valued validated Byzantine agreement below: Protocol VBA, described in Section 4.3, needs $O(n)$ rounds and invokes $O(n)$ binary agreement sub-protocols; this can be improved to a constant expected number of rounds, resulting in Protocol VBAconst, which is described in Section 4.4. But first we discuss the binary case.

4.2 Protocols for Binary Agreement

Binary asynchronous Byzantine agreement protocols can easily be adapted to external validity. For example, in the protocol of Cachin, Kursawe, and Shoup [7] one has to "justify" the pre-votes of round 1 with a valid π. The logic of the protocol guarantees that either a decision is reached immediately or the validations for 0 and for 1 are seen by all parties in the first two rounds.

Furthermore, the protocol can be biased towards 1 by modifying the coin such that it always outputs 1 in the first round.

4.3 A Protocol for Multi-valued Agreement

We describe Protocol VBA that implements multi-valued validated Byzantine agreement.

The basic idea of the validated agreement protocol is that every party proposes its value as a candidate value for the final result. One party whose proposal satisfies the validation predicate is then selected in a sequence of binary Byzantine agreement protocols and this value becomes the final decision value. More precisely, the protocol consists of the following steps (see Figure 3).

Echoing the proposal (lines 1–4): Each party P_i *c-broadcasts* the value that it proposes to all other parties using verifiable authenticated consistent broadcast. This ensures that all honest parties obtain the same proposal value for any particular party, even if the sender is corrupted. Then P_i waits until it has received $n - t$ proposals satisfying Q_{ID} before entering the agreement loop.

Agreement loop (lines 5–20): One party is chosen after another, according to a fixed permutation Π of $\{1, \ldots, n\}$. Let a denote the index of the party selected in the current round (P_a is called the "candidate"). Each party P_i carries out the following steps for P_a:

1. Send a `v-vote` message to all parties containing 1 if P_i has received P_a's proposal (including the proposal in the vote) and 0 otherwise (lines 6–11).
2. Wait for $n-t$ `v-vote` messages, but do not count votes indicating 1 unless a valid proposal from P_a has been received—either directly or included in the `v-vote` message (lines 12–13).
3. Run a *binary* validated Byzantine agreement biased towards 1 to determine whether P_a has properly broadcast a valid proposal. Vote 1 if P_i has received a valid proposal from P_a and validate this by the protocol message that completes the verifiable broadcast of P_a's proposal. Otherwise, if P_i has received $n-t$ `v-vote` messages containing 0, vote 0; no validation data is needed here. If the agreement decides 1, exit from the loop (lines 14–20).

Delivering the chosen proposal (lines 21–24): If P_i has not yet *c-delivered* the broadcast by the selected candidate, obtain the proposal from the validation returned by the Byzantine agreement.

The full protocol is shown in Figure 3.

An obvious optimization of Protocol VBA is based on the observation that in most cases, adding P_a's proposal in ρ to a `v-vote` message is not necessary. If this is omitted, then the code for P_i to receive `v-vote` messages has to be modified as follows. If a `v-vote` from P_j indicates 1 but P_i has not yet received P_a's proposal, ignore the vote and ask P_j to supply P_a's proposal (by sending it the message $(ID|\texttt{vcbc}.a.0, \texttt{c-request})$). The `v-vote` by P_j is only taken into account after $(ID, \texttt{v-echo}, w_a, \pi_a)$ has been *c-delivered* with tag $ID|\texttt{vcbc}.a.0$ such that $Q_{ID}(w_a, \pi_a)$ holds; however, it may still be that enough votes indicating 0 from other parties are received before that.

Lemma 7. *In Protocol* VBA*, the adversary can cause at most* $2t$ *iterations of the agreement loop.*

Proof. The proof works by counting the total number A of `v-vote` messages containing 0 that are generated by honest parties (over all iterations of the agreement loop).

Since every honest party has received a valid proposal from $n-t$ parties in the `v-echo` broadcasts, it will generate `v-vote` messages containing 0 for at most t proposing parties. Thus, $A \leq t(n-t)$.

Note that for the binary Byzantine agreement protocol to decide 0 for a particular a and to cause one more iteration of the loop, at least $n-2t$ honest parties must propose 0 for the binary agreement (otherwise, there would be $t+1$ or more honest parties proposing 1 and the binary agreement protocol would terminate with 1, as it is biased towards 1). Since honest parties only propose 0 if they have received $n-t$ `v-vote` messages containing 0, there must be at least $n-2t$ honest parties who have generated a `v-vote` message containing 0 in this iteration.

Let R denote the number of iterations of the loop where the binary agreement protocol decides 0. From the preceding argument, we have $A \geq R(n-2t)$.

Combining these two bounds on A, we obtain $R(n-2t) \leq (n-t)t$, or equivalently,

$$R \;\leq\; t + \frac{t^2}{n-2t}.$$

Using $n-2t \geq t+1$, this can be simplified to $R \leq t + \frac{t^2}{t+1}$ and further to $R < 2t$. Thus, the binary agreement decides 1 at the latest in iteration $R+1$ of the loop and the lemma follows. □

Protocol VBA **for party** P_i**, tag** ID**, and validation predicate** Q_{ID}

LET $V_{ID|a}(v, \rho)$ BE THE FOLLOWING PREDICATE:

$$V_{ID|a}(v,\rho) \;\equiv\; (v = 0) \textbf{ or } \big(v = 1 \textbf{ and } \rho \text{ completes the verifiable authenticated c-broadcast of a message } (\texttt{v-echo}, w_a, \pi_a) \text{ with tag } ID.a.0 \text{ such that } Q_{ID}(w_a, \pi_a) \text{ holds}\big)$$

UPON RECEIVING MESSAGE $(ID, \texttt{in}, \texttt{v-propose}, w, \pi)$:

verifiably authenticatedly c-broadcast message $(\texttt{v-echo}, w, \pi)$ tagged with $ID|\texttt{vcbc}.i.0$
$w_j \leftarrow \perp; \pi_j \leftarrow \perp \qquad (1 \le j \le n)$
wait for $n - t$ messages $(\texttt{v-echo}, w_j, \pi_j)$ to be *c-delivered* with tag $ID|\texttt{vcbc}.j.0$ from distinct P_j such that $Q_{ID}(w_j, \pi_j)$ holds
$l \leftarrow 0$
repeat
  $l \leftarrow l + 1; a \leftarrow \Pi(l)$
  if $w_a = \perp$ **then**
    send the message $(ID, \texttt{v-vote}, a, 0, \perp)$ to all parties
  else
    let ρ be the message that completes the c-broadcast with tag $ID|\texttt{vcbc}.a.0$
    send the message $(ID, \texttt{v-vote}, a, 1, \rho)$ to all parties
  $u_j \leftarrow \perp; r_j \leftarrow \perp \qquad (1 \le j \le n)$
  wait for $n - t$ messages $(ID, \texttt{v-vote}, a, u_j, \rho_j)$ from distinct P_j such that $V_{ID|a}(u_j, \rho_j)$ holds
  if there is some $u_j = 1$ **then**
    $v \leftarrow 1;\ \rho \leftarrow \rho_j$
  else
    $v \leftarrow 0;\ \rho \leftarrow \perp$
  propose v validated by ρ for $ID|a$ in binary validated Byzantine agreement biased towards 1, with predicate $V_{ID|a}$
  wait for the agreement protocol to decide some b validated by σ for $ID|a$
until $b = 1$
if $w_a = \perp$ **then**
  use σ to complete the verifiable authenticated c-broadcast with tag $ID|\texttt{vcbc}.a.0$ and *c-deliver* $(ID, \texttt{v-echo}, w_a, \pi_a)$
output $(ID, \texttt{out}, \texttt{v-decide}, w_a, \pi_a)$
halt

Figure 3: Protocol VBA for multi-valued validated Byzantine agreement.

Theorem 8. *Given a protocol for biased binary validated Byzantine agreement and a protocol for verifiable authenticated consistent broadcast, Protocol* VBA *provides multi-valued validated Byzantine agreement for $n > 3t$.*

Proof. We have to establish *external validity*, *agreement*, *liveness*, and *efficiency*.

External validity follows because every honest party that proposes 1 in the agreement on party P_a has verified that Q_{ID} holds for w_a and π_a. Thus, by the standard validity condition for the binary Byzantine agreement, the decision is 0 if Q_{ID} does not hold.

For *agreement*, note that the properties of the *binary* validated Byzantine agreement protocol ensure that all parties terminate the loop with the same a. By the consistency property of consistent broadcast, all honest parties obtain the same values w_a and π_a from the broadcast tagged with $ID|\texttt{vcbc}.a.0$. Thus, they output the same w_a.

Liveness and *integrity* hold by inspection of the protocol.

Efficiency follows from Lemma 3 together with Lemma 7 because there are at most $2t$ binary agreement sub-protocols invoked for a particular ID. □

The message complexity of Protocol VBA is $O(tn^2)$ if Protocol VCBC is used for verifiable consistent broadcast and the binary validated Byzantine agreement is implemented according to Section 4.2.

If all parties propose v and π that are together no longer than L bits, the communication complexity in the above case is $O(n^2(tK + L))$, assuming the length of a threshold signature and a signature share is at most K bits. For a constant fraction of corrupted parties, however, both values are cubic in n. As shown next, the expected message complexity can be reduced to a quadratic expression in n.

4.4 A Constant-round Protocol for Multi-valued Agreement

In this section we present Protocol VBAconst, which is an improvement of the protocol in the previous section that guarantees termination within a constant expected number of rounds. The drawback of Protocol VBA above is that the adversary knows the order Π in which the parties search for an acceptable candidate, i.e., one that has broadcast a valid proposal. Although at least one third of all parties are guaranteed to be accepted, as shown above, the adversary can choose the corruptions and schedule messages such that none of them is examined early in the agreement loop.

The remedy for this problem is to choose Π randomly during the protocol *after* making sure that enough parties are already committed to their votes on the candidates. This is achieved in two steps. First, one round of commitment exchanges is added before the agreement loop. Each party must commit to the votes that it will cast by broadcasting the identities of the $n - t$ parties from which it has received valid `v-echo` messages (using at least authenticated consistent broadcast). Honest parties will later only accept `v-vote` messages that are consistent with the commitments made before. The second step is to determine the permutation Π using a threshold coin-tossing scheme that outputs a random, unpredictable value after enough votes are committed. Taken together, these steps ensure that the fraction of parties which are guaranteed to be accepted are distributed randomly in Π, causing termination in a constant expected number of rounds.

The details of Protocol VBAconst are described in Figure 4 as modifications to Protocol VBA.

To analyze the protocol, we consider the state of the system at the point in time when the first honest party P_i reveals its coin share. The crucial observation is that $n - t$ "early committing" parties are committed to their 0-votes at this point because P_i has delivered the

Protocol VBAconst **for party P_i, tag ID, and validation predicate Q_{ID}**

Modify Protocol VBA for party P_i, tag ID, and validation predicate Q_{ID} as follows:

1. Initialize and distribute the shares for an $(n, t+1)$-threshold coin-tossing scheme $\mathcal{C}_1$ with k''-bit outputs during system setup. Recall that this defines a pseudorandom function F. Let G be a pseudorandom generator according to Section 2.3.

2. Include the following instructions between lines 3 and 4 of Protocol VBA, before entering the agreement loop:

   $c_j \leftarrow \begin{cases} 1 & \text{if } w_j \neq \perp \\ 0 & \text{otherwise} \end{cases} \qquad (1 \leq j \leq n)$

   $C \leftarrow [c_1, \ldots, c_n]$

   authenticatedly c-broadcast the message $(\texttt{v-commit}, C)$ tagged with $ID|\texttt{cbc}.i.0$

   $C_j \leftarrow \perp \qquad (1 \leq j \leq n)$

   wait for $n - t$ messages $(\texttt{v-commit}, C_j)$ to be *c-delivered* with tag $ID|\texttt{cbc}.j.0$
   such that at least $n - t$ entries in C_j are 1

   generate a *coin share* γ of the coin $ID|\texttt{vba}$ and send the message $(ID, \texttt{v-coin}, \gamma)$
   to all parties

   wait for $t + 1$ `v-coin` messages containing shares of the coin $ID|\texttt{vba}$ and
   combine these to get the value $S = F(ID|\texttt{vba}) \in \{0,1\}^{k''}$

   choose a random permutation Π, using the pseudorandom generator G with seed S.

3. Modify the condition for accepting `v-vote` messages (line 13) inside the agreement loop such that $(\texttt{v-vote}, a, 0, \perp)$ from P_j is accepted only if C_j is known and $C_j[a] = 0$. (This involves also waiting for additional messages $(\texttt{v-commit}, C_j)$ to be *c-delivered* as above.)

Figure 4: Protocol VBAconst for multi-valued validated Byzantine agreement.

corresponding broadcasts. We are now going to investigate the number of candidates that can be rejected by the adversary, by making the binary Byzantine agreement decide 0, and the number of iterations of the agreement loop.

Lemma 9. *Let $\mathcal{A} \subseteq \{1, \ldots, n\}$ denote the set of parties that garner* less *than $n - 2t$ commitments to 0-votes from the early committers, and suppose Π is an ideal, random permutation of $\{1, \ldots, n\}$. Then, except with negligible probability,*

1. *for every $a \in \mathcal{A}$, the binary agreement protocol on $ID|a$ will decide 1;*
2. *$|\mathcal{A}| > n - 2t$;*
3. *there exists a constant $\beta > 1$ such that for all $f \geq 1$,*

$$\Pr\Big[\big(\Pi(1) \notin \mathcal{A}\big) \wedge \cdots \wedge \big(\Pi(f) \notin \mathcal{A}\big)\Big] \;\leq\; \beta^{-f}.$$

Proof. In order for the binary agreement for $ID|a$ to decide 0, there must be some honest party who proposes 0. By the instructions for computing v, it must have received $n - t$ `v-vote` messages containing 0 that are consistent with the commitments made by their issuers. But since there are only n distinct parties, at least $n - 2t$ of those 0-votes must come from early committers, which is not the case for any $a \in \mathcal{A}$. This proves the first claim.

To establish the second claim, let A denote the total number of commitments to 0-votes cast by early committers. Since every early committer may commit to voting 0 for at most t parties, we have $A \leq t(n - t)$. On the other hand, observe that $A \geq (n - |\mathcal{A}|)(n - 2t)$ by the definition of $\mathcal{A}$.

Observe that these bounds on A are the same as in Lemma 7 with $R = n - |\mathcal{A}|$. Using the same argument, it follows $|\mathcal{A}| > n - 2t$.

The third claim follows now because $|\mathcal{A}|$ is at least a constant fraction of n and thus, there is a constant $\beta > 1$ such that $\Pr[\Pi(i) \notin \mathcal{A}] \leq 1/\beta$ for all $1 \leq i \leq f$. Since the probability of the f first elements of Π jointly satisfying the condition is no larger than for f independently and uniformly chosen values, we obtain

$$\Pr\Big[\big(\Pi(1) \notin \mathcal{A}\big) \wedge \cdots \wedge \big(\Pi(f) \notin \mathcal{A}\big)\Big] \;\leq\; \beta^{-f}.$$

□

Lemma 10. *Assuming C_1 is a secure threshold coin-tossing scheme and G is a pseudorandom generator, there is a constant $\beta > 1$ such that for all $f \geq 1$, the probability of any honest party performing f or more iterations of the agreement loop is at most $\beta^{-f} + \epsilon$, where ϵ is negligible.*

Proof. This can be shown by a standard hybrid argument, where one makes a series of small modifications to transform an idealized system into the real system, argues that each change affects the adversary only with negligible probability, and then concludes that the real system behaves just like the idealized system with all but negligible probability.

The "hybrid systems" are defined by running the system

(1) with a truly random permutation Π,

(2) with the output of G replaced by truly random bits, and Π computed from that,

(3) with $F(ID|\texttt{vba})$ replaced by a random bit string, but G being a pseudorandom generator according to the protocol, and Π computed from the output of G,

(4) with F, G, and Π computed according to the protocol.

In all cases, we define a statistical test by letting the adversary run the system until the first honest party is about to release its share of the coin $ID|\texttt{vba}$, and then F, G, and Π are determined. Note that the set of early committers is defined and the set $\mathcal{A}$ (of Lemma 9) can be computed at this point. The statistical test simply outputs 0 if $\Pi(i) \notin \mathcal{A}$ for all $1 \leq i \leq f$ and 1 otherwise.

We now analyze the behavior of the statistical test.

Case (1) above corresponds to the idealized system in Lemma 9, which implies that the test outputs 0 at most with probability β^{-f}.

In case (2) above, the permutation is generated from truly random bits with uniform distribution. This can be done using an algorithm that always terminates in a polynomial number of steps such that the output permutation is statistically close to a random permutation. The behavior of any polynomial-time adversary will not be changed by this, except with negligible probability.

Cases (2) and (3) above can be mapped to the definition of a pseudorandom generator. But if G is secure, the statistical test will not be able to distinguish between them with more than negligible probability.

Finally, the difference between (3) and (4) corresponds to game C1–C4 in the definition of the coin F. Assuming F is pseudorandom, this cannot induce more than a negligible difference in the behavior of the statistical test.

In conclusion, we obtain that no polynomial-time statistical test can distinguish between (1) and (4) and therefore the conclusions of Lemma 9 apply also to the real protocol except with negligible probability. Since honest parties go through more than f iterations of the agreement loop only if the first f elements of Π are not in $\mathcal{A}$, this probability is at most β^{-f} plus some negligible quantity. □

Theorem 11. *Given a protocol for biased binary validated Byzantine agreement and a protocol for verifiable consistent broadcast, Protocol* VBAconst *provides multi-valued validated Byzantine agreement for $n > 3t$ and invokes a constant expected number of binary Byzantine agreement sub-protocols.*

Proof. Since we have not changed the way in which binary agreement sub-protocols are invoked from Protocol VBA, we only have to show *liveness* and *efficiency* for the modified protocol.

Liveness holds because all $n-t$ honest parties broadcast correctly constructed commitments and therefore, enough valid `v-commit` and `v-vote` messages are guaranteed to be received in line 13 of the original protocol.

Efficiency follows from Lemma 3 together with Lemma 10 above, because honest parties generate a polynomial number of messages in each iteration of the agreement loop. □

The expected message complexity of Protocol VBAconst is $O(n^2)$ if Protocol VCBC is used for consistent verifiable broadcast and the binary validated Byzantine agreement is implemented according to Section 4.2.

If all parties propose v and π that are together no longer than L bits, the expected communication complexity in the above case is $O(n^3 + n^2(K+L))$, assuming a digital signature is K bits. The n^3-term, which results from broadcasting the commitments, has actually a very small hidden constant because the commitments can be represented as bit vectors.

For a constant fraction of corrupted parties, the message complexity is quadratic in n and essentially optimal. We do not know whether the communication complexity can be lowered to a quadratic expression in n as well.

5 Atomic Broadcast

Atomic broadcast guarantees a total order on messages such that honest parties deliver all messages with a common tag in the same order. It is well known that protocols for atomic broadcast are considerably more expensive than those for reliable broadcast because even in the crash-fault model, atomic broadcast is equivalent to consensus [10] and cannot be solved by deterministic protocols. The atomic broadcast protocol given here builds directly on multi-valued validated Byzantine agreement from the last section.

5.1 Definition

Atomic broadcast ensures that all messages broadcast with the same tag ID are delivered in the same order by honest parties; in this way, ID can be interpreted as the name of a broadcast "channel." The total order of atomic broadcast yields an implicit labeling of all messages. Assuming some honest party has atomically delivered s distinct messages, the global sequence of the first s delivered messages is well-defined. Thus, an explicit sequence number is not needed. Since the sender of a payload message is not necessarily identifiable (without requiring explicit authenticity instead of integrity), the sender name is also omitted, and an unstructured tag ID suffices.

An atomic broadcast is activated when the adversary delivers an input message to P_i of the form

$$(ID, \texttt{in}, \texttt{a-broadcast}, m),$$

where $m \in \{0,1\}^*$. When this occurs, we say P_i *atomically broadcasts* m *with tag* ID. "Activation" here refers only to the broadcast of a particular payload message; the broadcast channel ID must be opened before the first such request.

A party terminates an atomic broadcast of a particular payload by generating an output message of the form

$$(ID, \texttt{out}, \texttt{a-deliver}, m).$$

In this case, we say P_i *atomically delivers* m *with tag* ID. To distinguish atomic broadcast from other forms of broadcast, we will also use the terms *a-broadcast* and *a-deliver*.

For the composition of atomic broadcast with other protocols, we need a synchronized output mode, where *a-delivering* a payload may block the protocol and prevent it from delivering more payloads until the consumer is ready to accept them. We introduce an acknowledgment mechanism for output messages for this purpose, i.e., the adversary should *acknowledge* every *a-delivered* payload message to the delivering party. In practice, the *a-delivery* operation could be implemented by a blocking upcall to the higher-level protocol. In terms of the formal model, an acknowledgment is modeled as an input message $(ID, \texttt{in}, \texttt{a-acknowledge})$ from the adversary. When a party receives such a message, it means that its most recently *a-delivered* payload message with tag ID has been *acknowledged.* We will say that the adversary *generates acknowledgments* if it acknowledges every *a-delivered* message.

Again, the adversary must not request an *a-broadcast* of the same payload message from any particular party more than once for each ID (however, several parties may *a-broadcast* the same message).

Atomic broadcast protocols should be *fair* so that a payload message m is scheduled and delivered within a reasonable (polynomial) number of steps after it is *a-broadcast* by an honest party. But since the adversary may delay the sender arbitrarily and *a-deliver* an a priori unbounded number of messages among the remaining honest parties, we can only provide such a guarantee when at least $t+1$ honest parties become "aware" of m. Our definitions of validity and of fairness require actually that only after $t+1$ honest parties have *a-broadcast* some payload, it will be delivered within a reasonable number of steps. This is also the reason for allowing multiple parties to *a-broadcast* the same payload message—a client application might be able to satisfy this precondition through external means and achieve guaranteed fair delivery in this way. Fairness can be interpreted as a termination condition for the broadcast of a particular payload m.

The *efficiency* condition (which ensures fast termination) for atomic broadcast differs from the protocols discussed so far because the protocol for a particular tag cannot terminate on its own. It merely stalls if no more undelivered payload messages are in the system and must be terminated externally. Thus, we cannot define efficiency using the absolute number of protocol messages generated. Instead we measure the progress of the protocol with respect to the number of messages that are *a-delivered* by honest parties. In particular, we require that the number of associated protocol messages does not exceed the number of *a-delivered* payload messages times a polynomial factor, independent of the adversary.

We say that a protocol message is *associated* to the atomic broadcast protocol with tag ID if and only if the message is generated by an honest party and tagged with ID or with a tag $ID|\ldots$ starting with ID. In particular, this encompasses all messages of the atomic broadcast protocol with tag ID generated by honest parties and all messages associated to basic broadcast and Byzantine agreement sub-protocols invoked by atomic broadcast.

Fairness and efficiency are defined using the number of payload messages in the "implicit queues" of honest parties. We say that a payload message m is *in the implicit queue of a party P_i (for channel ID)* if P_i has *a-broadcast* m with tag ID, but *no honest party* has *a-delivered* m tagged with ID. The *system queue* contains any message that is in the implicit queue of *some* honest party. We say that one payload message in the implicit queue of an honest party P_i is *older* than another if P_i *a-broadcast* the first message before it *a-broadcast* the second one.

When discussing implicit queues at particular points in time, we consider a sequence of events $E_1, \ldots, E_{k'''}$ during the operation of the system, where each event but the last one is either an *a-broadcast* or *a-delivery* by an honest party. The phrase "at time τ" for $1 \leq \tau \leq k'''$ refers to the point in time just *before* event E_τ occurs.

Definition 7 (Atomic Broadcast). A protocol for *atomic broadcast* satisfies the following conditions except with negligible probability:

Validity: There are at most t honest parties with non-empty implicit queues for some channel ID, provided the adversary opens channel ID for all honest parties, delivers all associated messages, and generates acknowledgments.

Agreement: If some honest party has *a-delivered* m tagged with ID, then all honest parties *a-deliver* m tagged with ID, provided the adversary opens channel ID for all honest parties, delivers all associated messages, and generates acknowledgments for every party that has not yet *a-delivered* m tagged with ID.

Total Order: Suppose an honest party P_i has *a-delivered* $m_1, \ldots, m_s$ with tag ID, a distinct honest party P_j has *a-delivered* $m'_1, \ldots, m'_{s'}$ with tag ID, and $s \leq s'$. Then $m_l = m'_l$ for $1 \leq l \leq s$.

Integrity: For all ID, every honest party *a-delivers* a payload message m at most once tagged with ID. Moreover, if all parties follow the protocol, then m was previously *a-broadcast* by some party with tag ID.

Fairness: Fix a particular protocol instance with tag ID. Consider the system at any point in time τ_0 where there is a set $\mathcal{T}$ of $t+1$ honest parties with non-empty implicit queues, let $\mathcal{M}$ be the set consisting of the oldest payload message for each party in $\mathcal{T}$, and let S_0 denote the total number of distinct payload messages *a-delivered* by any honest party so far. Define a random variable W as follows: let W be the total number of distinct payload messages *a-delivered* by honest parties at the point in time when the first message in $\mathcal{M}$ is *a-delivered* by any honest party, or let $W = S_0$ if this never occurs. Then $W - S_0$ is uniformly bounded.

Efficiency: For a particular protocol instance with tag ID, let X denote its communication complexity, and let Y be the total number of distinct payload messages that have been *a-delivered* by any honest party with tag ID. Then, at any point in time, the random variable $X/(Y+1)$ is probabilistically uniformly bounded.

Some remarks on the above definition:

1. Compared to the definition of reliable broadcast, agreement and integrity are analogous, validity is somewhat weaker, and total order and fairness are new.

2. The agreement condition combines the consistency and totality of reliable broadcast; there is no need to distinguish these two aspects here. However, only totality requires that messages and acknowledgments are delivered.

3. Validity ensures liveness of a protocol and rules out trivially empty protocols. It is stated in a weak form, guaranteeing progress whenever at least $t+1$ honest parties have some undelivered payload message. A stronger notion, more along the lines of the validity condition used in reliable broadcast, would have been the following.

   Strong Validity: If an honest party has *a-broadcast* m tagged with ID, then it *a-delivers* m tagged with ID, provided the adversary opens channel ID for all honest parties, delivers all associated messages, and generates acknowledgments.

   However, our weaker notion of validity is sufficient in many applications where a client needs to contact more than $t+1$ servers anyway. It is also more suitable for protocol composition and makes some atomic broadcast protocols simpler, like the one of Kursawe and Shoup [23]. On the other hand, strong validity can be obtained for any atomic broadcast protocol that provides weak validity by a relatively simple initial round of echoing the payload to all parties, who adopt it as their own if their input queues are empty.

4. Validity and fairness complement each other: Validity ensures that a payload message that is *a-broadcast* by $t+1$ honest parties is *a-delivered* at all, provided all messages are delivered and acknowledgments are generated, and fairness implies that it is *a-delivered* reasonably quickly, relative to other payloads.

   One could define a weaker versions of fairness and validity by considering only the situation that f honest parties *a-broadcast* a payload message for $t+1 \leq f \leq n-t$.

5. The efficiency condition counts only the payload messages delivered by the "fastest" honest party. This party will usually be synchronized within one round with at least $n-2t-1$ other honest parties, but it seems impossible to synchronize it with the "slowest" honest

party. Moreover, there seems to be no easy way to provide a fixed bound on a suitable statistic (such as communication complexity) until *all* honest parties have delivered a particular payload. This is because the adversary can always drive the system forward with only $n - 2t$ honest parties and leave the others behind. The "fast" parties might generate an a priori unbounded amount of work until the "slow" ones finally *a-deliver* a particular payload, if at all. (Adding 1 to the divisor covers the state until the first payload is delivered.)

5.2 A Protocol for Atomic Broadcast

We now present a protocol for atomic broadcast based on validated Byzantine agreement. Its overall structure is similar to the protocol of Hadzilacos and Toueg [21] for the crash-fault model, but we need to take additional measures to tolerate Byzantine faults.

Our Protocol ABC for atomic broadcast proceeds as follows. Each party maintains a FIFO queue of not yet *a-delivered* payload messages. Messages received to *a-broadcast* are appended to this queue whenever they are received. The protocol proceeds in asynchronous global rounds, where each round r consists of the following steps:

1. Send the first payload message w in the current queue to all parties, accompanied by a digital signature σ in an `a-queue` message.

2. Collect the messages of $n - t$ distinct parties and store them in a vector W, store the corresponding signatures in a vector S, and propose W for Byzantine agreement validated by S.

3. Perform multi-valued Byzantine agreement with validation of a vector $W = [w_1, \dots, w_n]$ and proof $S = [\sigma_1, \dots, \sigma_n]$ through the predicate $Q_{ID|\mathtt{abc}.r}(W, S)$ which is true if and only if for at least $n-t$ distinct indices j, the vector element σ_j is a valid $\mathcal{S}$-signature on $(ID, \mathtt{a\text{-}queue}, r, j, w_j)$ by P_j.

4. After deciding on a vector V of messages, deliver the union of all payload messages in V according to a deterministic order; proceed to the next round.

In order to ensure liveness of the protocol, there are actually two ways in which the parties move forward to the next round: when a party receives an *a-broadcast* input message (as stated above) and when a party receives an `a-queue` message of another party pertaining to the current round. If either of these two messages arrive and contain a yet undelivered payload message, and if the party has not yet sent its own `a-queue` message for the current round, then it enters the round by appending the payload to its queue and sending an `a-queue` message to all parties.

The detailed description of Protocol ABC is found in Figure 5. The FIFO queue q is an ordered list of values (initially empty). It is accessed using the operations *append*, *remove*, and *first*, where *append*(q, m) inserts m into q at the end, *remove*(q, m) removes m from q (if present), and *first*(q) returns the first element in q. The operation $m \in q$ tests if an element m is contained in q.

A party waiting at the beginning of a round simultaneously **waits for `a-broadcast`** and `a-queue` messages containing some $w \notin d$ in line 2. If it receives an *a-broadcast* request, the payload m is appended to q. If only a suitable `a-queue` protocol message is received, the party makes w its own message for the round, but does not append it to q. It should be clear from the protocol that no honest party is ever blocked waiting for some payload message to process if some honest party has *a-broadcast* one and all associated messages have been delivered.

Protocol ABC **for party** P_i **and tag** ID

LET $Q_{ID|\mathtt{abc}.r}$ BE THE FOLLOWING PREDICATE:

$$Q_{ID|\mathtt{abc}.r}([w_1,\ldots,w_n],[\sigma_1,\ldots,\sigma_n]) \equiv \big(\text{for at least } n-t \text{ distinct } j,\ \sigma_j \text{ is a valid } \mathcal{S}\text{-signature by } P_j \text{ on } (ID, \mathtt{a\text{-}queue}, r, j, w_j).\big)$$

INITIALIZATION:

$q \leftarrow []$ {FIFO queue of messages to *a-broadcast*}
$d \leftarrow \emptyset$ {set of *a-delivered* messages}
$r \leftarrow 0$ {current round}

UPON RECEIVING MESSAGE $(ID, \mathtt{in}, \mathtt{a\text{-}broadcast}, m)$:

if $m \notin d$ **and** $m \notin q$ **then**
  $append(q, m)$

FOREVER:

$w_j \leftarrow \perp; \sigma_j \leftarrow \perp$ $(1 \leq j \leq n)$
wait for $q \neq []$ **or** a message $(ID, \mathtt{a\text{-}queue}, r, l, w_l, \sigma_l)$ received from P_l
  such that $w_l \notin d$ and σ_l is a valid signature from P_l
if $q \neq []$ **then**
  $w \leftarrow first(q)$
else
  $w \leftarrow w_l$
compute a digital signature σ on $(ID, \mathtt{a\text{-}queue}, r, i, w)$
send the message $(ID, \mathtt{a\text{-}queue}, r, i, w, \sigma)$ to all parties
wait for $n-t$ messages $(ID, \mathtt{a\text{-}queue}, r, j, w_j, \sigma_j)$ such that σ_j is a valid
  signature from P_j (including the message from P_l above)
$W \leftarrow [w_1,\ldots,w_n]; S \leftarrow [\sigma_1,\ldots,\sigma_n]$
propose W validated by S for multi-valued validated Byzantine agreement
  for $ID|\mathtt{abc}.r$ with predicate $Q_{ID|\mathtt{abc}.r}$
wait for the validated Byzantine agreement protocol to decide some
  $V = [v_1,\ldots,v_n]$ for $ID|\mathtt{abc}.r$
$b \leftarrow \bigcup_{j=1}^{n} v_j$
for $m \in (b \setminus d)$, in some deterministic order **do**
  output $(ID, \mathtt{out}, \mathtt{a\text{-}deliver}, m)$
  wait for an acknowledgment
  $d \leftarrow d \cup \{m\}$
  $remove(q, m)$
$r \leftarrow r + 1$

Figure 5: Protocol ABC for atomic broadcast using multi-valued validated Byzantine agreement.

The term $n - t$ in line 9 of the protocol and in the validation predicate $Q_{ID|\mathsf{abc}.r}$ could be replaced by any f' between $t + 1$ and $n - t$ if the fairness condition is changed such that $f = n - f' + 1$ parties instead of $t + 1$ must have *a-broadcast* the message.

The protocol in Figure 5 is formulated using a single loop that runs forever after initialization; this is merely for syntactic convenience and can be implemented by decomposing the loop into the respective message handlers.

Theorem 12. *Given a protocol for multi-valued validated Byzantine agreement and assuming $\mathcal{S}$ is a secure signature scheme, Protocol* ABC *provides atomic broadcast for $n > 3t$.*

Proof. We first prove *validity* and show that the protocol even implements strong validity. Towards a contradiction, suppose that some honest party has *a-broadcast* a payload message m, but not *a-delivered* it and yet, all associated protocol messages and acknowledgments have been delivered. Since the sender has *a-broadcast* but not *a-delivered* m, its queue q contains at least m and it can no longer be waiting in line 2. Thus, it has proceeded and sent `a-queue` messages to all parties in line 8. Since these have been delivered, every honest party has received an `a-queue` message containing $m \notin d$ and therefore has also entered the same round (by condition for waiting in line 2). Thus, all $n - t$ honest parties have sent valid `a-queue` messages and every honest party has received all of them and subsequently started and terminated Byzantine agreement. Since also the *a-delivered* payloads have been acknowledged, the sender must be waiting in line 2 with $q = []$. But then m has been removed from q and this occurs only if it was *a-delivered*, a contradiction.

We now establish *agreement*. Towards a contradiction, suppose that some honest P_i has *a-delivered* a payload message m, but an honest P_j has not *a-delivered* it and yet, all associated protocol messages have been delivered and acknowledgments have been generated for all parties who have not yet *a-delivered* m. Assume P_i *a-delivered* m in round r. Since no party who has not *a-delivered* m is blocked waiting for messages or acknowledgments under these conditions, it is easy to see from inspection of the protocol and from the liveness condition of the Byzantine agreement sub-protocol that P_j must have received all messages belonging to any round up to and including r. But then it cannot be waiting for an acknowledgment either—unless it has already *a-delivered* m.

The *total order* condition follows from the agreement property of the validated Byzantine agreement primitive since all honest parties decide on the same proposal and then *a-deliver* all payload messages contained in the proposal in a deterministic order. This implies also that the set d of *a-delivered* messages is the same for all honest parties.

Integrity is immediate from the protocol by induction on the construction of d, using the properties of Byzantine agreement. Even if corrupted parties include messages that have already been delivered, they are not delivered again.

To show *fairness*, fix some τ_0 and $\mathcal{T}$ (this defines also $\mathcal{M}$), and consider the system at the point in time when $W > 0$ is first defined. We show that $W - S_0 \leq n$, independent of the adversary. Note that the decided vector in the current round is defined and contains $n - t$ payloads (not necessarily distinct). At least $n - 2t$ of them are signed by honest parties that have all caught up to the current round; we call these parties the "signing parties." They have each signed the oldest payload message in their queue q. By definition, the implicit queue of every honest party is a subset of q; but because each signing party must have entered the current round, its implicit queue was *equal* to its queue q at the point in time when it generated the signature. Since $\mathcal{T}$ has cardinality $t + 1$ and there are at least $n - 2t$ signing parties, but only $n - t$ honest parties in total, there must be at least one signing party in $\mathcal{T}$. Thus, there is at least one payload from $\mathcal{M}$ among the decided payloads and no more than n distinct payloads can have been *a-delivered* since τ_0.

For *efficiency*, we have to relate the communication complexity of the protocol to the payload messages that were actually *a-delivered.* Note that honest parties generate messages only when they make progress in the round structure—either by sending an `a-queue` message or by invoking the Byzantine agreement sub-protocol. But an honest party enters the next round only if it is aware of some payload message that it has not yet *a-delivered.* Since at least one payload message from the system queue is delivered in every round, all protocol messages generated during that round can be related to that payload. There are a fixed polynomial number of protocol messages generated directly by the protocol in every round and the length of each one is at most n times the length of a payload. The communication complexity of the Byzantine agreement sub-protocol is probabilistically uniformly bounded by its efficiency condition. Thus, the communication complexity per round is probabilistically uniformly bounded. □

The message complexity of Protocol ABC to broadcast one payload message m is dominated by the number of messages in the multi-valued validated Byzantine agreement; the extra overhead for atomic broadcast is only $O(n^2)$ messages. The same holds for the communication complexity, but the proposed values have length $O(n(|m|+K))$, assuming digital signatures of length K bits.

With Protocol VBAconst from Section 4.4, the total expected message complexity is $O(n^2)$ and the expected communication complexity is $O(n^3|m|)$ for an atomic broadcast of a single payload message.

5.3 Equivalence of Byzantine Agreement and Atomic Broadcast

For the sake of completeness, we state the equivalence of atomic broadcast to Byzantine agreement in the cryptographic model. It is the analogue to the equivalence between consensus and atomic broadcast in the crash-fault model shown by Chandra and Toueg [10].

Corollary 13. *(Binary) Byzantine agreement and atomic broadcast are equivalent in the basic system model of Section 2.1, assuming a secure signature scheme and provided $n > 3t$.*

Proof. To implement Byzantine agreement from an atomic broadcast protocol, a party uses the following algorithm:

1. To propose $v \in \{0,1\}$ for transaction ID, compute a digital signature σ on (ID, v) and *a-broadcast* the message (ID, v, σ).

2. Wait for *a-delivery* of the first $2t+1$ messages of the form (ID, v_j, σ_j) from distinct parties that contain valid signatures. Decide for the simple majority of all received values v_j.

The other direction follows from Theorems 8 and 12. □

Note that using an appropriately defined notion of *authenticated* atomic broadcast, this could also be implemented without the additional digital signatures in the reduction. However, Protocol ABC would have to be modified in order to provide authentication.

6 Secure Causal Atomic Broadcast

Secure causal atomic broadcast (SC-ABC) is a useful protocol for building secure applications that use state machine replication in a Byzantine setting. It provides atomic broadcast, which ensures that all recipients receive the same sequence of messages, and also guarantees that the payload messages arrive in an order that maintains "input causality," a notion introduced by Reiter and Birman [39]. Informally, input causality ensures that a Byzantine adversary may not ask the system to deliver any payload message that depends in a meaningful way on a yet undelivered payload sent by an honest client. This is very useful for delivering client requests to a distributed service in applications that require the contents of a request to remain secret until the system processes it. Input causality is related to the standard causal order (going back to Lamport [24]), which is a useful safety property for distributed systems with crash failures, but is actually not well defined in the Byzantine model [21].

Input causality can be achieved if the sender encrypts a message to broadcast with the public key of a threshold cryptosystem for which all parties share the decryption key [39]. The ciphertext is then broadcast using an atomic broadcast protocol; after delivering it, all parties engage in an additional round to recover the message from the ciphertext.

In our description of secure causal atomic broadcast, one of the parties acts as the sender of a payload message. If SC-ABC is used by a distributed system to broadcast client requests, then encryption and broadcasting is taken care of by the client. In this case, additional considerations are needed to ensure proper delivery of the replies from the service (see [39] for those details).

6.1 Definition

Associated with any instance of a secure causal atomic broadcast protocol with tag ID is an encryption algorithm E_{ID}. It should be possible to infer this algorithm from the dealer's public output. E_{ID} is a probabilistic algorithm that maps a message m to a ciphertext c. We call $c = E_{ID}(m)$ an encryption of m (with tag ID). Since the encryption algorithm is probabilistic, there will in general be many different encryptions of a given message; indeed, this will necessarily be the case if the system is to be secure.

An application that wants to securely broadcast a payload message should first encrypt it using E_{ID} and invoke the broadcast protocol with the resulting ciphertext. Since E_{ID} is publicly known, also clients from outside the group $P_1, \ldots, P_n$ can produce ciphertexts.

A secure causal atomic broadcast protocol is activated when P_i receives an input message of the form

$$(ID, \texttt{in}, \texttt{s-broadcast}, c).$$

We say P_i *s-broadcasts* c with tag ID.

Unlike atomic broadcast, delivery consists of two distinct steps: the first is the generation of an output message of the form

$$(ID, \texttt{out}, \texttt{s-schedule}, c),$$

and the second is the generation of an output message of the form

$$(ID, \texttt{out}, \texttt{s-reveal}, m).$$

We shall require that honest parties generate sequences of such pairs of output messages—there must never be two consecutive `s-schedule` or `s-reveal` messages. When the `s-schedule` message is generated, we will say that P_i *s-schedules* the ciphertext c (with tag ID). When the `s-reveal` message is generated, we will say that P_i *s-delivers* the ciphertext c (with tag ID), where c is the most recently *s-scheduled* ciphertext; we call m the *associated cleartext.*

Definition 8 (Secure Causal Atomic Broadcast). A secure causal atomic broadcast protocol satisfies the properties of an atomic broadcast protocol, where the *s-broadcast* and *s-delivery* of ciphertexts in the secure causal atomic broadcast protocol play the role of the *a-broadcast* and *a-delivery* of payload messages in an atomic broadcast protocol.

Additionally, the following conditions hold.

Message Secrecy: According to the basic system model, the parties run an atomic broadcast protocol (and possibly other broadcast protocols), and the adversary plays the following game:

B1. The adversary interacts with the honest parties in an arbitrary way.

B2. The adversary chooses two messages m_0 and m_1 and a tag ID; it gives them to an "encryption oracle." The oracle chooses a bit B at random and computes an encryption c of m_B with tag ID, and gives this ciphertext to the adversary.

B3. The adversary continues to interact with the honest parties subject only to the condition that no honest party *s-schedules* c with tag ID.

B4. Finally, the adversary outputs a bit $\hat{B}$.

Then, for any adversary, the probability that $\hat{B} = B$ must exceed $\frac{1}{2}$ only by a negligible amount.

Message Integrity: According to the basic system model, the parties run an atomic broadcast protocol (and possibly other broadcast protocols), and the adversary plays the following game:

C1. The adversary interacts with the honest parties in an arbitrary way.

C2. The adversary chooses a message m and a tag ID, and gives it to an "encryption oracle." The oracle computes an encryption c of m with tag ID, and gives this ciphertext to the adversary.

C3. The adversary continues to interact with the honest parties in an arbitrary way.

We say the adversary wins the game if at some point an honest party *s-delivers* c with tag ID, but corresponding cleartext m' is not equal to m. Then, for any adversary, the probability that it wins this game is negligible.

Message Consistency: If two parties honest parties *s-deliver* the same ciphertext c with tag ID, then with all but negligible probability, the associated cleartexts are the same.

It is easy to verify that this definition implies input causality in the sense of Reiter and Birman [39], i.e., that a cleartext remains hidden from the adversary until the corresponding ciphertext is *s-scheduled.* But the cleartext may be revealed to the adversary before the first honest party outputs it in a `s-reveal` message, and this is also the reason for introducing our two-step delivery process. Although this is necessary for the proper definition of security, *s-scheduling* a ciphertext might be omitted in a practical implementation.

The *message integrity* condition gives clients access to the broadcast protocol for cleartext payload messages, and implies that payloads contained in correctly encrypted ciphertexts are actually output by the honest parties.

6.2 A Protocol for Secure Causal Atomic Broadcast

Protocol SC-ABC in Figure 6 implements secure causal atomic broadcast. It uses an $(n, t+1)$-threshold cryptosystem $\mathcal{E}_1$ that is secure against adaptive chosen ciphertext attacks (see Section 2.3.3) for which the parties share the decryption key. It also uses an atomic broadcast protocol according to Section 5.

During initialization, the dealer generates a public key for $\mathcal{E}_1$, together with the corresponding private key shares, and distributes them according to the initialization algorithm of $\mathcal{E}_1$.

For a tag ID, $E_{ID}(m)$ is computed by applying the encryption algorithm of $\mathcal{E}_1$ to m with label ID, using the generated public key of the cryptosystem.

We emphasize that all instances of the secure causal broadcast protocol share the same public key for $\mathcal{E}_1$, and so the use of *labeled ciphertexts* is essential to properly "isolate" different instances of the protocol from one another.

To *s-broadcast* a ciphertext c, we simply *a-broadcast* c. Upon *a-delivery* of a ciphertext c, a party *s-schedules* c. Then it computes a decryption share δ and sends this to all other parties in an `s-decrypt` message containing c. It waits for $t+1$ `s-decrypt` messages pertaining to c. Once they arrive, it recovers the associated cleartext and *s-delivers* c. After receiving the acknowledgment, the party continues processing the next *a-delivery* by generating the corresponding acknowledgment. The details are in Figure 6. For ease of notation, the protocol in Figure 6 is formulated using a Forever loop; it can be decomposed into the respective message handlers in straightforward way.

Protocol SC-ABC **for party** P_i **and tag** ID

Initialization:

open an atomic broadcast channel with tag $ID|$`scabc`

Upon receiving $(ID, \texttt{in}, \texttt{s-broadcast}, c)$:

a-broadcast c with tag $ID|$`scabc`

Forever:

wait for the next message c that is *a-delivered* with tag $ID|$`scabc`
compute an $\mathcal{E}_1$-decryption share δ for c with label ID
output $(ID, \texttt{out}, \texttt{s-schedule}, c)$
send the message $(ID, \texttt{s-decrypt}, c, \delta)$ to all parties
$\delta_j \leftarrow \perp \qquad (1 \le j \le n)$
wait for $t+1$ messages $(ID, \texttt{s-decrypt}, c, \delta_j)$ from distinct parties that contain valid decryption shares for c with label ID
combine the decryption shares $\delta_1, \ldots, \delta_n$ to obtain a cleartext m
output $(ID, \texttt{out}, \texttt{s-reveal}, m)$
wait for an acknowledgment
acknowledge the last *a-delivered* message with tag $ID|$`scabc`

Figure 6: Protocol SC-ABC for secure causal atomic broadcast.

Theorem 14. *Given an atomic broadcast protocol and assuming $\mathcal{E}_1$ is a $(n, t+1)$-threshold cryptosystem secure against adaptive chosen-ciphertext attacks, Protocol* SC-ABC *provides secure causal atomic broadcast for $n > 3t$.*

Proof. We have to show that the protocol implements atomic broadcast and satisfies message secrecy and message integrity conditions.

We first show *validity.* Suppose enough honest parties have *s-broadcast* c and all associated messages have been delivered and all acknowledgments have been generated. Thus, all senders have *a-broadcast* c. We can now invoke the validity condition of the atomic broadcast protocol as follows: first, the messages associated to the atomic broadcast have been delivered since they are also associated to the secure broadcast; second, it is clear from the protocol that the acknowledgements to the secure broadcast protocol are passed on to the atomic broadcast protocol. Thus, the validity of the atomic broadcast implies that c has been *a-delivered* by some honest party. For the same reasons, the agreement condition of atomic broadcast implies that all other honest parties must also have *a-delivered* c, since they are not blocked or waiting for acknowledgements. All honest parties have therefore generated decryption shares for c and sent an `s-decrypt` message to all parties. It follows that any honest party has received at least $t+1$ valid shares for c. But then it has also *s-delivered* c.

It is perhaps interesting to note that the above proof of *validity* made essential use of *both* the *validity* and *agreement* properties of the underlying atomic broadcast protocol.

For *agreement*, suppose that an honest P_i has *s-delivered* c and P_j has not, and yet, all associated messages have been delivered and acknowledgments have been generated for those parties who have not *s-delivered* c. Since any honest party that has not yet *s-delivered* c has received sufficiently many acknowledgements, it has also acknowledged all *a-deliveries* and it cannot be waiting for an acknowledgment in the atomic broadcast protocol. Since P_i has *a-delivered* c, it follows from the agreement condition of the underlying atomic broadcast that all other honest parties must also have *a-delivered* c. Thus, they all have generated decryption shares for c and P_j must have received at least $t+1$ valid shares for c. Therefore, P_j has *s-delivered* c, a contradiction.

To show *efficiency*, we must bound the amount of work done (as measured by communication complexity) per *s-delivered* message. But since the *s-delivery* messages is synchronized with the *a-delivery* of ciphertexts in Protocol SC-ABC, the number of *a-delivered* messages exceeds the number of *s-delivered* ones by at most one, and efficiency follows from the efficiency condition of the atomic broadcast protocol.

Note that without this synchronization, we could not achieve efficiency, since the lower-level atomic broadcast protocol could "run ahead" of the higher-level secure causal atomic broadcast protocol—lots of messages would be generated, but very few messages would be *s-delivered.*

It is easy to see that the remaining broadcast properties (*total order*, *integrity*, and *fairness*) hold as well, using the corresponding properties of the underlying atomic broadcast.

Message secrecy, integrity, and consistency follow easily from the properties of the underlying threshold encryption scheme. □

7 Conclusions

Although cryptographic techniques play an important role in the development of secure fault-tolerant systems, the formal methods used in cryptography and in distributed systems seem rather different today. An integration of both approaches, such as the one proposed in this paper, is therefore desirable for developing secure distributed protocols.

Apart from the definitions, this paper presents several new protocols for asynchronous broadcast and Byzantine agreement problems. They illustrate how fault-tolerant broadcasts can benefit from threshold-cryptographic protocols such as threshold signatures and coin-tossing. In particular, they lead to improved communication complexity. Our most efficient protocol for atomic broadcast achieves $O(n^2)$ expected message complexity to broadcast a single payload message and expected communication complexity $O(n^3)$.

Several interesting problems remain open:

– Our corruption model is static, i.e., the adversary must decide which parties to corrupt independently from the behavior of the system. Allowing for adaptive corruptions would give stronger security guarantees, but it is currently not known how to efficiently realize all of our threshold-cryptographic primitives with adaptive security.

– Although the communication complexity per payload message of the atomic broadcast protocol seems reasonable for relatively small values of n, it would be nice to reduce it further to $O(n^2)$, or even to a smaller expression. This improvement would have to be made in the multi-valued validated Byzantine agreement protocol.

Another approach for reducing the overhead of atomic broadcast in practice are dual-mode protocols, which normally operate in a fast "optimistic" mode, and only switch to a slower "pessimistic" mode if no progress seems to be made during a certain time. The protocol of Castro and Liskov [9] is of this type, but it does not guarantee liveness in a fully asynchronous model. Recently, Kursawe and Shoup [23] have developed such an "optimistic" atomic broadcast protocol that guarantees liveness and safety at the same time, and exploits many of the techniques developed in this work.

Acknowledgments

This work was supported by the European IST Project MAFTIA (IST-1999-11583). However, it represents the view of the authors. The MAFTIA project is partially funded by the European Commission and the Swiss Department for Education and Science.

References

[1] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proc. 1st ACM Conference on Computer and Communications Security*, 1993.

[2] M. Ben-Or, R. Canetti, and O. Goldreich, "Asynchronous secure computation," in *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, 1993.

[3] M. Ben-Or, B. Kelmer, and T. Rabin, "Asynchronous secure computation with optimal resilience," in *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC)*, 1994.

[4] P. Berman and J. A. Garay, "Randomized distributed agreement revisited," in *Proc. 23th International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 412–419, 1993.

[5] G. Bracha, "An asynchronous $[(n-1)/3]$-resilient consensus protocol," in *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 154–162, 1984.

[6] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM*, vol. 32, pp. 824–840, Oct. 1985.

[7] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography," in *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 123–132, 2000. Full version available from Cryptology ePrint Archive, Report 2000/034, `http://eprint.iacr.org/`.

[8] R. Canetti and T. Rabin, "Fast asynchronous Byzantine agreement with optimal resilience," in *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 42–51, 1993. Updated version available from `http://www.research.ibm.com/security/`.

[9] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. Third Symp. Operating Systems Design and Implementation (OSDI)*, 1999.

[10] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[11] S. Chaudhuri, "More choices allow more faults: Set consensus problems in totally asynchronous systems," *Information and Computation*, vol. 105, no. 1, pp. 132–158, 1993.

[12] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, pp. 644–654, Nov. 1976.

[13] D. Dolev, C. Dwork, and M. Naor, "Non-malleable cryptography," *SIAM Journal on Computing*, vol. 30, no. 2, pp. 391–437, 2000.

[14] D. Dolev and H. R. Strong, "Authenticated algorithms for Byzantine agreement," *SIAM Journal on Computing*, vol. 12, pp. 656–666, Nov. 1983.

[15] A. Doudou, B. Garbinato, and R. Guerraoui, "Abstractions for devising Byzantine-resilient state machine replication," in *Proc. 19th Symposium on Reliable Distributed Systems (SRDS 2000)*, pp. 144–152, 2000.

[16] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *Foundations of Computation Theory* (M. Karpinsky, ed.), vol. 158 of *Lecture Notes in Computer Science*, Springer, 1983. Also published as Tech. Report YALEU/DCS/TR-273, Department of Computer Science, Yale University.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, pp. 374–382, Apr. 1985.

[18] O. Goldreich, *Foundations of Cryptography: Basic Tools.* Cambridge University Press, 2001. To appear.

[19] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *Journal of the ACM*, vol. 33, pp. 792–807, Oct. 1986.

[20] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal on Computing*, vol. 17, pp. 281–308, Apr. 1988.

[21] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems* (S. J. Mullender, ed.), New York: ACM Press & Addison-Wesley, 1993. An expanded version appears as Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[22] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing protocols for securing group communication," in *Proc. 31st Hawaii International Conference on System Sciences*, pp. 317–326, IEEE, Jan. 1998.

[23] K. Kursawe and V. Shoup, "Optimistic asynchronous atomic broadcast." Cryptology ePrint Archive, Report 2001/022, Mar. 2001. `http://eprint.iacr.org/`.

[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[25] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July 1982.

[26] N. Lynch and M. R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proc. 6th ACM Symposium on Principles of Distributed Computing (PODC)*, 1987.

[27] N. A. Lynch, *Distributed Algorithms.* San Francisco: Morgan Kaufmann, 1996.

[28] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quaterly*, vol. 2, pp. 219–246, Sept. 1989.

[29] D. Malkhi, M. Merritt, and O. Rodeh, "Secure reliable multicast protocols in a WAN," *Distributed Computing*, vol. 13, no. 1, pp. 19–28, 2000.

[30] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography.* Boca Raton, FL: CRC Press, 1997.

[31] L. E. Moser and P. M. Melliar-Smith, "Byzantine-resistant total ordering algorithms," *Information and Computation*, vol. 150, pp. 75–111, 1999.

[32] M. Naor, B. Pinkas, and O. Reingold, "Distributed pseudo-random functions and KDCs," in *Advances in Cryptology: EUROCRYPT '99* (J. Stern, ed.), vol. 1592 of *Lecture Notes in Computer Science*, Springer, 1999.

[33] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, pp. 228–234, Apr. 1980.

[34] D. Powell (Guest Ed.), "Group communication," *Communications of the ACM*, vol. 39, pp. 50–97, Apr. 1996.

[35] M. O. Rabin, "Randomized Byzantine generals," in *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 403–409, 1983.

[36] M. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in Rampart," in *Proc. 2nd ACM Conference on Computer and Communications Security*, 1994.

[37] M. K. Reiter, "The Rampart toolkit for building high-integrity services," in *Theory and Practice in Distributed Systems*, vol. 938 of *Lecture Notes in Computer Science*, pp. 99–110, Springer, 1995.

[38] M. K. Reiter, "Distributing trust with the Rampart toolkit," *Communications of the ACM*, vol. 39, pp. 71–74, Apr. 1996.

[39] M. K. Reiter and K. P. Birman, "How to securely replicate services," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 986–1009, May 1994.

[40] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

[41] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, Dec. 1990.

[42] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology: EUROCRYPT 2000* (B. Preneel, ed.), vol. 1087 of *Lecture Notes in Computer Science*, pp. 207–220, Springer, 2000.

[43] V. Shoup and R. Gennaro, "Securing threshold cryptosystems against chosen ciphertext attack," in *Advances in Cryptology: EUROCRYPT '98* (K. Nyberg, ed.), vol. 1403 of *Lecture Notes in Computer Science*, Springer, 1998.

[44] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev, "Group communication specifications: A comprehensive study," Technical Report MIT-LCS-TR-790, MIT, Sept. 1999.