In Proceedings of CRYPTO-87, ed. C. Pomerance, 1988, pp. 379-391

How to Make Replicated Data Secure

Maurice P. Herlihy and J. D. Tygar
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Many distributed systems manage some form of long-lived data, such as files or data bases. The performance and fault-tolerance of such systems may be enhanced if the repositories for the data are physically distributed. Nevertheless, distribution makes security more difficult, since it may be difficult to ensure that each repository is physically secure, particularly if the number of repositories is large. This paper proposes new techniques for ensuring the security of long-lived, physically distributed data. These techniques adapt replication protocols for fault-tolerance to the more demanding requirements of security. For a given threshold value, one set of protocols ensures that an adversary cannot ascertain the state of a data object by observing the contents of fewer than a threshold of repositories. These protocols are cheap; the message traffic needed to tolerate a given number of compromised repositories is only slightly more than the message traffic needed to tolerate the same number of failures. A second set of protocols ensures that an object's state cannot be altered by an adversary who can modify the contents of fewer than a threshold of repositories. These protocols are more expensive; to tolerate t-1 compromised repositories, clients executing certain operations must communicate with t-1 additional sites.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order Numbers 4864 and 4976, monitored by the Air Force Avionics Laboratory under Contracts F33615-84-K-1520 and N00039-84-C-0467.

1. Introduction

A *distributed system* consists of a collection of computers, called sites, that are geographically distributed and connected by a communications network. Examples of distributed systems include distributed inventory control systems, banking systems, airline reservation systems, and campus-wide file systems. Many distributed systems manage some form of long-lived data, such as files or databases. In this paper, we propose new techniques for ensuring the security of long-lived data in distributed systems. Our notion of security encompasses two properties: (1) *secrecy*, ensuring that unauthorized parties cannot observe confidential data (e.g., by stealing a disk pack), and (2) *integrity*, ensuring that unauthorized parties cannot corrupt or modify valuable data (e.g., by doctoring a disk pack).

Physical distribution makes security more difficult. In a centralized system, perhaps the most straightforward approach to security is to keep long-lived data in a physically secure location (e.g., keep the disk drive in a locked room.) When repositories for data are physically distributed, however, it is more difficult to ensure that each one is physically secure. As the number of sites increases, so does the number of ways in which the secrecy and integrity of the data can be compromised.

A similar difficulty arises when attempting to guarantee a distributed system's *availability*: the larger the number of independently-failing components, the smaller the likelihood they will all be working simultaneously, and the smaller the likelihood the system will be accessible when needed. This well-known phenomenon is typically addressed by designing distributed systems to be *fault-tolerant*, that is, able to function correctly in the presence of some number of failures. In particular, the availability of long-lived data can be enhanced by storing the data redundantly at multiple sites, a technique commonly known as *replication* [7, 11].

In this paper, we consider how replication protocols originally proposed to enhance fault-tolerance can be adapted to the more demanding requirements of security. For a given *threshold* value t, we describe and analyze several encryption-based secrecy protocols that ensure that an adversary cannot ascertain the object's state by observing the contents of fewer than t repositories. We then extend these protocols to guarantee integrity, ensuring that the object's state cannot be altered by an adversary who can tamper with the contents of fewer than t repositories. To keep our presentation as straightforward as possible, we assume that a communications subsystem provides secure and authenticated communication using known protocols, e.g., [2, 14, 10, 8]. Here, we consider only attacks that bypass the communications subsystem, isolating some set of repositories and directly observing or modifying their data.

A natural basis for analyzing the costs of our protocols is to compare the cost of replication for availability (tolerating t failures) with the cost of replication for security (tolerating t compromised sites). We find that:

- *Secrecy is cheap.* We propose two alternative secrecy protocols, one based on private key encryption, and one based on public key encryption. The private key scheme has the advantage that encryption itself is fast, but

a client may sometimes have to communicate with additional sites the first time it operates on an object. The public key scheme is more flexible, but known public key encryption algorithms are slower than their private key counterparts. We believe that the private key encryption method could be practical for real systems since the cost of security is dominated by the cost of replication.

- *Integrity is expensive.* To tolerate *t-1* compromised repositories, clients executing certain operations must communicate with *t-1* additional sites. We show that our protocol is optimal in the sense that any protocol that requires less communication is vulnerable to a simple "spoofing" attack.

A naive solution, at least for security, is simply to issue encryption keys to clients, storing only encrypted data at vulnerable repositories. A weakness of this approach is that it does not address key management. If the client keeps the key in non-volatile storage, then an adversary can compromise the security of the data simply by compromising the client's local storage. If the client keeps the key in volatile memory, a key management service is necessary both to initialize clients, and to redistribute new keys when the data is reencrypted. Care must be taken that the key management service itself does not become a security risk (what happens if a site belonging to the key management service becomes compromised?) or an availability bottleneck (what happens if the key management service's sites are down?). The protocols described in this paper address the security and availability aspects of key management by integrating key distribution directly into the replication protocols.

In Section 2 we define some terminology and we review two algorithms that form the basis for our protocols: quorum consensus replication and shared secrets. In Section 3 we describe a protocol employing private key encryption to preserve the secrecy of data against a passive adversary, and in Section 4 we describe an alternative secrecy protocol employing public key encryption. Section 5 discusses protocols for on-the-fly reencryption. In Section 6, we describe a protocol that preserves the integrity of the data against an active adversary, and Section 7 concludes with a survey of related work and a brief discussion.

2. Background

2.1. Terminology

We use two notions of cryptographic security in this paper. One notion, of *bit-security*, [3, 19], implies that given ciphertext, no processor with randomized polynomial resources can derive information about any given bit in the corresponding cleartext with certainty greater than $1/2+\varepsilon$ for any $\varepsilon > 0$. Given the current limits of complexity theory, we do not have a way of proving bit-security without making some assumption about the lower bound on an algorithmic problem. For example, it has been shown [1] that the RSA [20] cryptosystem is bit-secure under the assumption that taking k^{th} roots modulo pq, a product of two large primes, cannot be done in randomized polynomial time and

that the Rabin [15] signature schemes is bit-secure if factorization is not in randomized polynomial time. (These assumptions are generally accepted by the academic computer science community.)

The second notion, of *perfect security*, implies that given ciphertext, no processor with unlimited resources can derive a probability distribution of the corresponding cleartext other than a uniform distribution. For example, Shamir's secret sharing method, described in Section 2.3, satisfies the requirements of perfect security; the range of secrets compatible with fewer than a threshold number of pieces ranges over the entire set of integers in the finite field. Clearly, a cryptographic method that has perfect security is also bit-secure. When we say that a ciphertext gives *no information* about the corresponding cleartext, we mean that it satisfies either of the above definitions of security.

2.2. Quorum Consensus Replication

A complete description of quorum consensus replication is given elsewhere [11]. For brevity, we give two informal examples: files and counters.

A replicated object is implemented by two kinds of modules: *repositories* and *front-ends*. Repositories provide long-term storage for the object's state, while front-ends carry out operations for clients. Different objects may have different sets of repositories. Because front-ends can be replicated to an arbitrary extent, perhaps placing one at each client's site, the availability of a replicated object is dominated by the availability of its repositories. A *quorum* for an operation is any set of repositories whose cooperation suffices to execute that operation. A *quorum assignment* associates each operation with a set of quorums. An operation's quorums determine its availability, and the constraints governing an object's quorum assignments determine the range of availability properties realizable by a replication method.

A replicated file [7] is represented as a collection of timestamped versions, where timestamps are generated by logical clocks [12]. To read a file, a front-end reads the version from a *read quorum* of repositories and chooses the version with the latest timestamp. To write the file, a front-end generates a new timestamp and records the newly timestamped version at a write quorum of repositories. A quorum assignment is correct if and only if each read quorum has a non-empty intersection with each write quorum.

File replication algorithms are unsuited for certain kinds of objects. Consider a *counter* object that assumes integer values. It provides three operations: *Inc()* increments the counter by one, *Dec()* decrements the counter by one, and *Value()* returns the counter's current value. Such a counter is used as part of a protocol for dynamic reconfiguration of replicated objects [11]. The counter is a replicated *reference count* that keeps track of the number of existing references to a particular replicated object. The counter is incremented whenever a new reference is created, and decremented whenever an existing reference is destroyed. The system periodically checks the counter's value,

reclaiming storage for objects whose counters have reached zero.

R1	R2	R3
1:01 Inc()		1:01 Inc()
1:02 Inc()	1:02 Inc()	
	2:03 Dec()	2:03 Dec()

Figure 2-1: A Replicated Counter

As shown in Figure 2-1, a replicated counter is represented as a partially replicated log of entries, where an entry is a timestamped record of an Inc or Dec operation. It is convenient to split an operation's quorums into two parts: an *initial* quorum and a *final* quorum. To increment or decrement the counter, the front-end simply sends a timestamped entry to a final quorum for the operation. To read the counter's value, the front-end merges the logs from a final quorum for the operation, using the timestamps to discard duplicates. A quorum assignment is correct if and only if (1) each final Dec quorum intersects each initial Value quorum, and (2) each final Inc quorum intersects each final Value quorum. Many other examples of replicated typed objects appear elsewhere [11].

2.3. Shared Secrets

In this section, we give a brief overview of Shamir's *secret sharing* algorithm [22]. This algorithm transforms a cleartext value v into n encrypted pieces such that any t pieces determine the value of v, but an adversary in possession of $t-1$ pieces has no information, in the sense defined in Section 2.1, about the value of v.

We do all computations with integer residues modulo a large prime p. Let the polynomial

$$h(x) = r_1x^{t-1} + r_2x^{t-2} + \ldots + r_{t-1}x + v$$

where the r_i are independent random integer residues. The pieces of the secret are the values $h(1), \ldots, h(n)$. The secret itself is $v=h(0)$. Since t distinct values of h determine a nonsingular linear system of t equations in t variables, Gaussian elimination will allow us to recover the values of the coefficients of h and hence v. On the other hand, since our operation is over a finite field, $t-1$ values of h yield no information about the value of v.

Creation of secrets require evaluation of a $t-1$-th degree polynomial (time $O(t)$) and recovery of the original text requires solution of a small linear system (time $O(t^3)$). (We are assuming here a fixed p; the complexity of these operations is polynomial in the logarithm of p.) This compares very favorably with standard private key and public key encryption techniques [13].

3. Private Key Secure Quorum Consensus

The protocol described here protects the secrecy of the object against an adversary who can observe the contents of fewer than *t* repositories. It depends on the existence of a bit-secure *private key* encryption scheme, in which a single key *K* is used for both encryption and decryption. The encryption scheme must be *probabilistic* [9] to ensure that repeated instances of the same cleartext (e.g., *Inc* entries for a replicated counter) produce different ciphertext instances.

3.1. Overview

The private key Secure Quorum Consensus (SQC) protocol is implemented by three kinds of sites:

1. *Front Ends* interact directly with clients, but have only volatile memory.
2. *Repositories* communicate only with other processors and have a long-term store.
3. A *Dealer* communicates only with other processors and has a source of random bits. The stores his data exclusively in volatile memory. The dealer could be a separate processor, but in general the dealer can be any one of a set of processors -- for example, the dealer can be one of the front ends or repositories. If one dealer becomes inaccessible, another processor can take over as the dealer, thus the dealer need not be a performance or availability bottleneck.

The SQC protocol consists of three phases:

1. *Object initialization:* The dealer chooses a random key *K*, and uses the shared secret protocol to divide *K* into *n* pieces such that any *t* pieces suffice to reconstruct *K*. One piece is sent to each of the *n* repositories for the object, and all data stored at the repositories will be encrypted with *K*
2. *Front-end initialization:* When a front-end comes on-line, it ascertains *K* by reading the pieces from *t* out of *n* repositories. The front-end records *K* in a volatile cache, which must be reinitialized on crash recovery, or when the object is reencrypted.
3. *Operation Execution:* The front-end executes a modified quorum consensus protocol. In the first step, it reads the encrypted data from an initial quorum of repositories. In the second step, it decrypts the data using *K*, performs the operation, and encrypts the new data using *K*.

Recall that all messages are assumed to be secure and authenticated by the communications subsystem.

The definition of a bit-secure private key cryptosystem and Shamir's secret sharing algorithm immediately imply:

Property 1: An adversary who reads the contents of fewer than *t* repositories can derive no information about the object's state.

Nevertheless, notice that an adversary may still be able to derive useful information from "traffic analysis," such as observing when an object's value changes or has been reencrypted. For example, if versions or log entries are timestamped in cleartext, then the amount of decryption needed to execute an operation can be reduced, but an adversary may be able to deduce information about the number and frequency of updates.

3.2. Examples

This section illustrates SQC by three examples representing three different availability/security trade-offs.

1. Consider a file for which the efficiency and availability of Read and Write operations are equally important. Here, read quorums, write quorums, and thresholds all require a majority $\lceil (n+1)/2 \rceil$ of repositories. Preserving secrecy incurs no additional cost in availability since a front-end can register at a quorum for its first operation. Up to $\lceil (n-1)/2 \rceil$ repositories may fail without loss of availability, and equally many repositories may be observed by an adversary without disclosing the contents of the file.

2. If the efficiency and availability of Read is more important, Read quorums could be defined to encompass any one repository, and Write quorums all *n*. Here, a threshold value of one is possible, but not prudent. Instead, likely threshold values range between two and a majority, trading the efficiency and availability of registration against the security of the data. Preserving secrecy incurs a one-time penalty, since a unregistered front-end cannot read the file from a read quorum. Nevertheless, the registration cost is amortized over the lifetime of the front-end.

3. At the other extreme, consider a replicated reference counter. The availability of the Inc and Dec operations is likely to be more important than the availability of the Value operation, since timely creation or destruction of references is more important than asynchronous garbage collection. Inc and Dec quorums could be defined to encompass any one repository, and Value quorums all *n*. Here, too, sensible threshold values range between two and a majority, and registration incurs a one-time availability penalty amortized over the lifetime of the front-end.

Preserving secrecy need not reduce availability if the threshold value can be set equal to the object's smallest quorum. If the smallest quorum consists of a single site, as in Examples 2 and 3, then private key SQC imposes a one-time registration penalty for each front-end. Clearly, some such penalty is inherent in Example 2, because if an unregistered front-end can ascertain the file's value from the data residing at a single

repository, so can an adversary. More surprisingly, the registration penalty in Example 3 can be eliminated, as we show in the next section.

4. Public Key Secure Quorum Consensus

In this section we describe a variant of SQC in which the bit-secure private key scheme is replaced by a bit-secure *public key* scheme [6]. Instead of a single key K, we use an *encryption key* K_E and a *decryption key* K_D, where K_D cannot be derived from K_E with polynomial resources, and vice-versa. Instead of a single threshold t, we use an *encryption threshold* t_E and a *decryption threshold* t_D. The dealer generates K_E and K_D, and uses the shared secret protocol to divide each one into n pieces such that any t_E pieces suffice to reconstruct K_E, and any t_D pieces suffice to reconstruct K_D. A front-end executing an operation uses K_D to decrypt the data it reads and K_E to encrypt the data it writes.

Because the encryption and decryption thresholds can be chosen independently, public key SQC is more flexible than its private key counterpart. For example, consider the replicated counter example from the previous section, in which quorums for Inc and Dec consist of any one repository, and a Value quorum consists of all n. By setting t_E to one and t_D to n, we enhance performance, availability, and security, eliminating the registration penalty, and permitting up to $n-1$ repositories to be compromised without disclosing the counter's value. If integrity is not a concern, there is no reason not to set t_E to one, effectively making K_E public. As discussed in Section 6, however, preserving integrity against an active adversary will require a higher value for t_E.

5. On-the-Fly Reencryption

To manage secure replicated data, it is important to be able to reencrypt data easily. A object might be reencrypted periodically to render useless any pieces of the key that have been inadvertently exposed, or an object might be reencrypted in response to suspicion that its current key has become compromised. In this section, we discuss how SQC supports reencryption. For brevity, we restrict our discussion here to files and private key SQC. We assume that the encrypted text includes enough internal redundancy (c.f., Section 6) that a front-end attempting to decrypt data with an out-of-date key will detect its mistake and reregister.

Consider a file replicated among n repositories having read quorums, write quorums, and thresholds of sizes r, w, and t.

> **Property 2:** A front-end that knows K can reencrypt the object with key K' as long as it has access to $\max(r,w,n-t+1)$ repositories.

The front-end requires access to a read quorum to read the file's current value, and write quorum to write out the reencrypted value. Suppose there are $t-1$ inaccessible repositories $R_1,\ldots,R_{t-1}$ whose pieces of the key are $s_1,\ldots,s_{t-1}$. The goal of the reencryption protocol is to choose a new key value K' which splits into pieces $s_1', \ldots, s_n'$ such that $s_i = s_i'$, for i between 1 and $t-1$. The front-end knows the values of the

polynomial coefficients K, r_1, ..., r_{t-1}, from which it can derive the values of the missing pieces $s_1,...,s_{t-1}$. To ensure that the new pieces agree with the old pieces at the missing repositories, the new polynomial's coefficients must satisfy a non-singular system of $t-1$ linear equations in t variables, thus the front-end is free to choose any value between 1 and p for K', and that choice determines all the other coefficients.

If p of the unaltered pieces are known to the adversary, however, then the threshold for the new key is effectively $t-p$ of $n-p$. If this level of security is unsatisfactory, then as long as $max(r,w,t)$ repositories are accessible, known protocols for on-the-fly reconfiguration (e.g., [11]) can be used to replace the inaccessible repositories with new, trusted repositories, using a completely new set of pieces to encrypt the key.

6. A Protocol for Preserving Integrity

So far, the protocols described here have been concerned with preserving the secrecy of data against a passive adversary. In this section we address the problem of preserving the *integrity* of data against an active adversary who can modify the local storage at fewer than t_I repositories. We wish to ensure that any such modification can be detected, implying that the compromised repository can be treated just as if it had crashed.

Our analysis is based on the following assumptions.

- The cleartext is encrypted together with a checksum that provides enough internal redundancy to detect any direct modifications to the ciphertext. Rabin and Karp have given such a checksum [21]. Another approach to this method is given by the more expensive technique of probabilistic encryption [9].
- The integrity threshold t_I is less than or equal to t (for private key SQC) or t_E (for public key SQC). Otherwise the problem is clearly hopeless.
- The adversary can, however, "spoof" the repository by taking a snapshot of its data, later replacing the current data with the out-of-date snapshot.

The basic idea is quite simple: We use either the public or private key SQC protocol, but instead of requiring quorum intersections to be non-empty, we require that quorum intersections have cardinality at least t_I. For files, this constraint ensures that each read quorum includes at least one uncompromised repository with the file's current value. Since the adversary is restricted to replaying old values with earlier timestamps, the version with the latest timestamp will be the correct one. In the counter example, each Value quorum will include at least one copy of each Inc and Dec entry.

This protocol may seem expensive, since at least one operation must have larger quorums, but the following argument (given here for files) shows that this protocol is optimal in the sense that any weaker constraint on quorum intersection yields a protocol vulnerable to spoofing. Consider a file replicated at n repositories, with read quorums of size r and write quorums of size w, where each read quorum intersects each write

quorum at x repositories: $r+w-x = n$. Let R, W, and X be disjoint sets of repositories of sizes $r-x$, $w-x$, and x, where the repositories in X are controlled by an adversary.

Consider the following scenario:

1. Client A initializes the file with value *a* at some write quorum.
2. The adversary snapshots the contents of each repository in X.
3. Client B writes the value *b* at the write quorum $W \cup X$.
4. At every repository in X, the adversary restores the snapshots taken in Step 2.
5. Client C reads the obsolete value *a* from the read quorum $R \cup X$.

In this scenario, B's write quorum intersects C's read quorum only at X, thus an adversary who controls X can spoof C into reading an obsolete value. Choosing quorums at random would provide a probabilistic guarantee of integrity by increasing the likelihood that a read quorum would intersect with the most recent write quorum at some uncompromised repository, but such a guarantee may well be too weak for applications where integrity is of concern.

7. Remarks and Related Work

Brian Randell [17, 18] posed the following question: can security and fault-tolerance be integrated in a single mechanism? This question challenges the traditional rule of thumb in security work that as a system becomes more distributed, vulnerability to security attacks increases. This paper answers Randell's question affirmatively, by giving a protocol in which security proportionally increases, with relatively low cost, as the system becomes more distributed. Unlike previous approaches, our protocol attacks the issues of security and fault-tolerance simultaneously with a single mechanism.

Tompa and Woll [23] have given a stronger version of the secret sharing protocol that is also applicable to our model. Chor *et al.* [4] have given a "verifiable secret sharing" protocol which ensures that the dealer cannot cheat. Chor and Rabin [5] have shown how this protocol can be used to generate bits with a high degree of independence in a Byzantine distributed system. The verifiable secret sharing protocol can also be incorporated into our techniques, but it would be more expensive to do so. The ITOSS [16] operating system used the secret sharing protocol to provide a distributed command processor for high security applications.

References

[1] W. Alexi, B. Chor, O. Goldreich, and C. P. Schnorr.
RSA and Rabin functions: certain parts are as hard as the whole.
In *Proceedings, 25th IEEE Symposium on the Foundations of Computer Science*. November, 1984.
To appear in SIAM J. on Computing.

[2] A. Birrell.
Secure communication using remote procedure calls.
ACM Transactions on Computer Systems 3(1):1-14, February, 1985.

[3] M. Blum and S. Micali.
How to generate cryptographically strong sequences of pseudo-random bits.
SIAM J. on Computing 13(4):850-864, 1984.

[4] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch.
Verifiable secret sharing and achieving simultaneity in the presence of faults.
In *Proceedings, 26th IEEE Symposium on the Foundations of Computer Science*. October, 1985.

[5] B. Chor and M. Rabin.
Achieving independence in a logarithmic number of rounds.
1987.
To appear, PODC 87.

[6] W. Diffie and M. E. Hellman.
New Directions in Cryptography.
IEEE Transactions on Information Theory IT-22(6):644-654, Nov., 1976.

[7] D. K. Gifford.
Weighted Voting for Replicated Data.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*. ACM SIGOPS, December, 1979.

[8] O. Goldreich, S. Micali, and A. Wigderson.
Proofs that yield nothing but the validity of their assertion.
Preprint.

[9] S. Goldwasser and S. Micali.
Probabilistic encryption and how to play mental poker.
In *Proceedings of the 14th Symposium on the Theory of Computation*. May, 1982.

[10] S. Goldwasser, S. Micali, and C. Rackoff.
The knowledge complexity of interactive proofs.
In *Proceedings of the 17th Symposium on the Theory of Computation*. May, 1985.

[11] M. P. Herlihy.
A quorum-consensus replication method for abstract data types.
ACM Transactions on Computer Systems 4(1), February, 1986.

[12] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558-565, July, 1978.

[13] C. Meyer and S. Matyas.
Cryptography.
Wiley, 1982.

[14] R. Needham and M. Schroeder.
Using encryption for authentication in large networks of computers.
Communications of the ACM 21(12), December, 1978.

[15] M. Rabin.
Digitalized signatures and public-key functions as intractable as factorization.
Technical Report MIT/LCS/TR-212, Laboratory for Computer Science, MIT, January, 1979.

[16] M. Rabin and J. D. Tygar.
An integrated toolkit for operating system security.
Technical Report TR-01-87, Aiken Computation Laboratory, Harvard University, January, 1987.

[17] B. Randell.
Recursively structured distributed computing systems.
In *Proceedings, Third Symposium on Reliability in Distributed Software and Database Systems*, pages 113-118. October, 1983.

[18] B. Randell and J. Dobson.
Reliability and security issues in distributed computing systems.
In *Proceedings, Fifth Symposium on Reliability in Distributed Software and Database Systems*, pages 113-118. January, 1986.

[19] J. Reif and J. D. Tygar.
Efficient parallel pseudo-random number generation.
In *Advances in Cryptology: CRYPTO-85*, pages 433-446. Springer-Verlag, August, 1985.
To appear in SIAM J. on Computing.

[20] R. Rivest, A. Shamir, and L. Adleman.
A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.
Communications of the ACM 21(2):120-126, Feb., 1978.

[21] Richard M. Karp and Michael O. Rabin.
Efficient Randomized Pattern-Matching Algorithms.
Technical Report TR 31-81, Aiken Laboratory, Harvard University, December, 1981.

[22] A. Shamir.
How to share a secret.
Communications of the ACM 22(11):612-614, Nov., 1979.

[23] M. Tompa and H. Woll.
How to share a secret with a cheater.
In *Advances in Cryptology: CRYPTO-86*. Springer-Verlag, 1986.
Also available as IBM Research Report RC-11840.