Optimistic Fair Exchange of Digital Signatures*

N. Asokan Victor Shoup Michael Waidner
IBM Zürich Research Laboratory, Säumerstr. 4, 8803 Rüschlikon, Switzerland
{aso,sho,wmi}@zurich.ibm.com

Abstract

We present a new protocol that allows two players to exchange digital signatures over the Internet in a fair way, so that either each player gets the other's signature, or neither player does. The obvious application is where the signatures represent items of value, for example, an electronic check or airline ticket. The protocol can also be adapted to exchange encrypted data. The protocol relies on a trusted third party, but is "optimistic," in that the third party is only needed in cases where one player attempts to cheat or simply crashes. A key feature of our protocol is that a player can always force a timely and fair termination, without the cooperation of the other player. A specialization of our protocol can be used for contract signing; this specialization is not only more efficient, but also has the important property that the third party can be held *accountable* for its actions: if it ever cheats, this can be detected and proven.

1 Introduction

As more business is conducted over the Internet, the *fair exchange problem* assumes increasing importance. For example, suppose player A is willing to give an electronic check to player B in exchange for an electronic airline ticket. The problem is this: how can A and B exchange these items so that either each player gets the other's item, or neither player does.

Both electronic checks and electronic airline tickets are implemented as digital signatures. Presumably, many other items to be exchanged over the Internet will be so implemented. Therefore, it seems fruitful to focus our attention on the fair exchange of digital signatures.

In this paper, we present a new protocol for the fair exchange of commonly used digital signatures, including RSA [20], DSS [15], Schnorr [21], Fiat-Shamir [10], GQ [13], and Ong-Schnorr [18] signatures, as well as the payment transcripts used in Brands' [5] off-line, anonymous cash scheme. The protocol can also be used to exchange data (such as stock quotes or an audio or visual stream) encrypted under a key derived from an RSA inverse or discrete logarithm. This scenario was suggested by Franklin and Reiter [11], where it was also suggested that the "quality" of the data be certified or guaranteed by an independent authority.

Our protocol uses a trusted third party, but only in a limited fashion: the third party is only needed in cases where one player attempts to cheat or simply crashes; therefore, in the vast majority of transactions, the third party will not need to be involved at all. Following [1], we call our protocol *optimistic*; in addition to [1], optimistic protocols for several variants of the fair exchange problem are discussed in [6, 17].

Of course, one could use an on-line trusted third party [8, 9, 12] in every transaction to act as a mediator, but the optimistic approach greatly reduces the load on the third party, which in turn

*This is IBM technical report RZ 2973, dated 11/17/97.

reduces the cost and insecurity involved in replicating the service in order to maintain availability. It also makes it more feasible to distribute the third party, eliminating the single point of failure.

Our protocol also enjoys the following properties:

(1) To use it, one need not modify the signature scheme or message format *at all*. Thus, it will inter-operate with existing or proposed schemes for electronic checks, coins, tickets, receipts, etc., without *any* modification to these schemes.

(2) It works in an asynchronous communication model: there are no synchronized clocks, and one player cannot force the other to wait for any length of time—a fair and timely termination can always be forced by contacting the third party.

(3) It is quite practical. A typical exchange requires only a few rounds of interaction, transmission of a few KBytes of data, and a couple thousand modular multiplications.

(4) The protocol can be proved secure (modulo standard intractability assumptions) in the random hash function model [3], where a hash function is replaced by a "black box" that outputs random bit strings.

(5) The two players may remain anonymous, if desired.

We stress the practical importance of property (1): it allows a general-purpose fair exchange service to be deployed without the cooperation of the institutions responsible for the items being exchanged (banks, airlines, etc.). Indeed, it seems quite unrealistic to expect these institutions to redesign their schemes and all of the relevant software to accommodate a fair exchange protocol if this has not already been designed for. Our protocol can accommodate any common signature scheme without modification. Previous optimistic protocols for fair exchange do not allow for this: these protocols either require that the item being exchanged have a special structure to facilitate the exchange protocol, or they partially sacrifice fairness, with one player ending up with just an affidavit from the third party that the other player owes him something. In our protocol, the two players get the real thing—not a substitute or affidavit.

Property (2) also needs to be stressed. Previous optimistic protocols for fair exchange could easily leave one player "hanging" for a long time, without knowing if the exchange was going to complete, and without being able to do anything about it.[1] Not only can this be a great inconvenience, it can also lead to a real loss in the case of time-sensitive data like stock quotes. In our protocol, this cannot happen so long as the third party is available.

We also point out that in practice, the most serious threat to a fair exchange is not malicious behavior by a player, but simply the possibility that one player crashes in the middle of the exchange. Our protocol deals equally well with both types of threats.

Contract Signing. In addition to the above general protocol for fair exchange of digital signatures, we give a specialized version of this protocol for contract signing. Here, we allow ourselves the additional flexibility of defining the form of a valid contract (sacrificing property (1) above). The protocol still works in an asynchronous communication model, is much more efficient than the general protocol, and enjoys the additional important property of *accountability:* if the trusted third party ever cheats, then this can be detected and proven. Thus, the third party can be held accountable for its actions, and perhaps be sued should it misbehave.

[1] Moreover, reducing the maximum "hang time" increases the risk of an unfair exchange.

1.1 Overview

At a high level, our protocol works as follows. Suppose the two players hold signatures on agreed-upon messages.

Each player first reduces a "promise" of a signature to "promise" of a particular homomorphic inverse (either a discrete logarithm or RSA inverse). A good reduction scheme should not make it any easier to compute the signature, while simultaneously guaranteeing that the promised signature can be recovered from the promised inverse. Good reduction schemes exist for all of the signature schemes mentioned above. While the reduction scheme for RSA is trivial, that for DSS is decidedly nontrivial, and may itself be of independent interest. The same notion was used by Franklin and Reiter [11] in the context of verifiably sharing a signature among several parties.[2]

After this reduction phase, the players perform a fair exchange of homomorphic inverses. As mentioned above, this sub-protocol can be used in exchanging data encrypted under a key derived (perhaps via a hash function) from a homomorphic inverse.

To exchange homomorphic inverses, our protocol makes use of "verifiable" encryptions of these inverses under the third party's public key: such an encryption can be verified to contain the homomorphic inverse, without leaking any information that makes it easier to compute the inverse. For this, we adapt Bellare and Goldwasser's [2] key escrow technique.[3]

Our protocol also employs a new strategy to deal with the asynchrony problem discussed above, allowing the exchange to be "aborted" under certain conditions.

The rest of the paper is organized as follows. In §2, we present a formal security model for the problem of exchanging digital signatures fairly. In §3, we show how to reduce the promise of a digital signature to the promise of a homomorphic inverse. In §4, we show how to verifiably encrypt a homomorphic inverse. In §5, we present our optimistic protocol for the fair exchange of digital signatures. In §6, we discuss a more efficient alternative to verifiable encryption that may be useful in certain situations. In §7, we present our specialized protocol for contract signing.

2 A Formal Security Model for Fair Signature Exchange

We have two players A and B, and a trusted third party T that acts as a server: it receives a request from a client, updates its internal state, and sends a response back to the client. T has a public key known to A and B, and is always assumed to be honest and (eventually) available. We assume the client/server channel is private, and that server responses are authenticated, but *do not* assume that the client requests are authenticated.[4]

The two players agree upon the signatures they want to exchange, and then exchange messages back and forth. Between the time that a player receives a message and generates its response, it may send requests to T, obtaining the corresponding responses within a finite, but unbounded, amount of time.

We define *security* in terms of *fairness* and *completeness.*

To define fairness, we let an adversary play the role of a corrupt player, and give it complete control over the network, arbitrarily interacting with T, and arbitrarily delaying A's requests to T. Intuitively, fairness means that it is infeasible for the adversary to get the honest player's signature, without the honest player getting the adversary's signature.

[2] A different reduction scheme for DSS is given in [11], but it is demonstrably insecure.

[3] For a different, but not sufficiently general approach, see [26].

[4] This can be implemented by having the client encrypt its request together with a session key under the third party's public key; the server encrypts and authenticates its response using the given session key; this implementation will have the desired properties provided the third party's encryption scheme is secure against chosen ciphertext attack.

Completeness means that if neither player is corrupt, and no messages are lost, then the exchange is successful.

We now make the above notions a bit more precise.

Behavior of T. T is a polynomial-time interactive Turing machine that follows the program prescribed for it by the protocol. T acts as a server, repeatedly accepting a request, updating its internal state, and generating a response. For simplicity, we assume that each request is processed *atomically*. T has a public key/private key pair (PK_T, SK_T) that is generated by a key generation algorithm prescribed by the protocol.

Behavior of an honest player. An honest player is a polynomial-time interactive Turing machine that follows the program prescribed for it by the protocol. It interacts with its environment through a sequence of rounds: in one round it receives a message, updates its internal state, and generates a response.

Before generating a response, it may access T (perhaps several times). To do this, the player must explicitly signal its intention to contact T, and then wait on an externally generated signal before proceeding.

The initial state of an honest player is determined by a set of inputs: $PK_T, PK, m, \sigma, PK', m'$. Here, σ is a signature on message m under the public key PK that the player intends to give in exchange for a signature on m' under the public key PK'.

After a bounded number of rounds, an honest player stops and writes on a private tape an output σ'. The player also externally signals that it has terminated.

Definition of fairness. Fix a particular signature scheme Σ, and consider the following game. The components in the game are an adversary, called B^*, which is a polynomial-time interactive Turing machine, an honest player, called A, as well as T, T's key generation algorithm, and Σ's key generation algorithm.

Game A

A1 Run Σ's key generation algorithm, giving the secret key to a signing oracle S and the public key PK to B^*. Also generate T's public and private keys, giving SK_T to T and PK_T to B^*.

A2 B^* interacts arbitrarily with T and S, obtaining signatures on adaptively chosen messages.

A3 B^* selects messages m and m', and an arbitrary public key PK' for a signature scheme (possibly different from Σ). The message m must be different from those given to S in A2. Now, S produces a signature σ on m, and A is initialized with inputs: $PK_T, PK, m, \sigma, PK', m'$. The signature σ is not seen by B^*.

A4 B^* interacts with T, S and A in an arbitrary fashion, subject to the following restrictions:

(1) B^* may not query S with m.

(2) When A signals its intention to contact T, B^* must eventually signal A to let it proceed, after which B^* refrains from contacting T until A is finished (i.e., generates a response or a signal).

(3) Unless A has signalled termination, B^* must eventually supply another input message.

Eventually, A terminates and outputs a string σ', and B^* also terminates, and outputs a string $\bar{\sigma}$. We say that B^* wins the game if σ' is not a valid signature for m' under PK', but $\bar{\sigma}$ is a valid signature for m under PK. We define *fairness* to mean that B^* cannot feasibly win this game.

Remarks. Restriction (2) captures our intuitive requirement that A can always reach T, but may be arbitrarily delayed. Despite restriction (3), real world "time outs" can be modeled quite simply: in the real world, if A is waiting for a message, then a low-level communication protocol will eventually time out and give A the message "?"; in our formal model, B^* just gives A this message directly. Also, note that in our definition, the player in the exchange protocol is not necessarily the holder of the signing key.

Definition of completeness. We define another game similar to that above, in which the adversary gets access to two signing oracles, S and S', and initializes two honest players A and B, who interact directly with each other. The adversary in this case can interact with T, but cannot interfere with the interaction of A and B, except insofar as the adversary still has the power to schedule both A's and B's interactions with T. We omit the details of this game, which are straightforward. The real-world situation that this game models is that where that all messages are delivered without being seen or modified by the adversary, and neither player "times out" waiting for a message.

We define *completeness* to mean that it is infeasible for the adversary in the above game to prevent A and B from successfully exchanging their signatures.

3 Reducing Signatures to Homomorphic Inverses

In this section, we show how to reduce a "promise" of a signature to the "promise" of a particular homomorphic inverse.

As a simple, motivating example, consider the standard hash-and-invert RSA signature (which is provably secure in the random hash function model [3]). Here, the signature on a message is simply the eth root of the hash of the message, viewed as a number modulo a given composite number. In this case, reduction is trivial: the promised homomorphic inverse is simply this eth root.

Reduction schemes for other signature schemes are not so trivial. So as to be able to uniformly treat several signature schemes, we start with a formal definition of our requirements.

3.1 Definition of a secure reduction

Let Σ be a signature scheme. A *reduction scheme* for Σ consists of three efficient algorithms, a *reduction algorithm* reduce, a *verification algorithm* verify, and a *recovery algorithm* recover, and also associates to every public key PK an efficiently computable group homomorphism $\theta : G_1 \to G_2$.

- reduce takes as input PK, m, σ, where PK is a public key for Σ, m is a message, and σ is signature on m under PK. The output consists of $d \in G_2$, $c \in \{0,1\}^*$, and $s \in \theta^{-1}(d)$. The string c encodes additional information that is used by the verification and recovery algorithms. reduce may fail on some "bad signatures," as long as these occur rarely (we need this for DSS).

- verify takes as input PK, m, d, c, and either accepts or rejects.

- recover takes as input PK, m, c, and $s \in G_1$, and outputs a string $\bar{\sigma}$.

A *secure* reduction scheme should satisfy three properties:

Completeness. If $\mathsf{reduce}(PK, m, \sigma) = (d, c, s)$, then $\mathsf{verify}(PK, m, d, c)$ accepts.

Soundness. It is infeasible for an adversary to find PK, m, d, c such that the $\mathsf{verify}(PK, m, d, c)$ accepts, but $\mathsf{recover}(PK, m, c, s)$ is not a valid signature on m for all $s \in \theta^{-1}(d)$.

Secrecy. It is infeasible for an adversary to win the following game:

Game B

B1 Run Σ's key generation algorithm, giving the secret key to a signing oracle S and the public key PK to the adversary.

B2 The adversary makes arbitrary queries to S.

B3 The adversary generates a message m different from those given to S in B2. Now S generates a signature σ on m under PK, and the adversary is given d, c, where $(d, c, s) = \mathsf{reduce}(PK, m, \sigma)$.

B4 The adversary continues to query S on messages different from m.

The adversary wins the game if it can output a valid signature on m.

Remarks. Clearly, our definition of a secure reduction scheme implies that the underlying signature scheme is secure against adaptive chosen-message attacks. A stronger definition could be formulated wherein the reduction scheme is "just as secure" as the underlying signature scheme, however secure it happens to be. Unfortunately, not all of our proofs achieve this. It is also possibly to consider more general reduction schemes; for example, the reduction procedure could be interactive. We do not consider these here.

3.2 Schnorr Signatures

In the Schnorr signature scheme, to generate a public key, one selects primes p and q such that $q \mid p-1$, and a generator g for the subgroup of $\mathbf{Z}_p^*$ of order q. One then chooses $x \in \mathbf{Z}_q$ at random and computes $h = g^x$. The public key consists of p, q, g, h, and the private key is x. To sign a message m, the signer chooses $r \in \mathbf{Z}_q$ at random, and computes $z = cx + r$, where $c = H(g^r, m) \in \mathbf{Z}_q$ and H is a hash function. The signature is (c, z). To verify a signature, one checks that $c = H(g^z h^{-c}, m)$.

The Schnorr signature scheme is provably secure in the random hash function model (if the discrete logarithm problem is hard).

The following reduction from Schnorr signatures to discrete logarithms was observed in [11], in the context of verifiable signature sharing. The reduction algorithm takes as input a signature (c, z) on a message m, and outputs $u = g^z, c$, where the promised inverse is $z = \log_g u$.

The verification algorithm checks that $c = h(uh^{-c}, m)$. Given z, the recovery algorithm outputs (c, z).

Completeness and soundness are clear. Secrecy follows from the fact that one can simulate the output of the reduction algorithm without a signature: the simulator chooses $r \in \mathbf{Z}_q$ at random, computes $w = g^r$ and $c = H(w, m)$, and sets $u = wh^c$.

3.3 DSS Signatures

Key generation for DSS is identical to that for the Schnorr scheme. The standard prescribes that q has a length of 160 bits. To sign a message m, the signer chooses $k \in \mathbf{Z}_q^*$ at random, and computes $r = (g^k) \bmod q$ and $s = k^{-1}(H(m) + xr)$, where H is a hash function with outputs in $\mathbf{Z}_q$. The signature is (r, s). A signature is verified by checking that $r = (g^{u_1} h^{u_2}) \bmod q$, where $u_1 = H(m)s^{-1}$, and $u_2 = rs^{-1}$. Note that signatures with $s = 0$ are invalid, but these effectively never arise. For our reduction scheme, we must also rule out $r = 0$, which also effectively never arises.

We now give a reduction scheme reducing a promise of a DSS signature to a promise of discrete logarithm.

The reduction algorithm works as follows. We have a signature (r, s) on a message m. Let u_1, u_2 be defined as above, and define $c = r^{-1}H(m)$. We output

$$\alpha = g^{u_1}, \beta = h^{u_2}, \alpha' = g^v, \beta' = h^v, z = v + eu_1,$$

where $v \in \mathbf{Z}_q$ is chosen at random, $e = H'(PK, \alpha, \beta, \alpha', \beta', c)$, and H' is a hash function.

The promised inverse is $u_2 = \log_h \beta$.

The verification algorithm runs as follows. We first check that $\alpha^q = \beta^q = 1$ and $\beta \neq 1$; we then compute $r = (\alpha\beta) \bmod q$, and check that $r \neq 0$, Finally, we compute $c = r^{-1}H(m)$, $e = H'(\alpha, \beta, \alpha', \beta', c)$, and check that $g^z = \alpha'\alpha^e$ and $h^z = \beta'\beta^{ce}$.

What is happening here is that we are giving α and β, and a non-interactive proof that $\log_g \alpha = c \log_h \beta$ (see [7]).

The recovery procedure takes $u_2 = \log_h \beta$, computes $r = (\alpha\beta) \bmod q$ and $s = r/u_2$, and outputs (r, s).

To prove secrecy, we have to make a *strong security assumption* about DSS: given signatures on several messages, not only is it difficult for an adversary to compute a signature on a new message, it is difficult to compute a *different* signature on any of the given messages.

One can heuristically justify this assumption using the "generic algorithm" model in [24], assuming also that H is collision free. Proving that DSS satisfies this stronger form of security in this model is relatively straightforward; moreover, this is the only model that we know of in which it is possible to prove that DSS is secure to begin with.

Lemma 1 *Under the strong security assumption for DSS, above reduction scheme is secure in the random hash function model for H'.*

Completeness. Clear.

Soundness. Let u_2 satisfying $h^{u_2} = \beta$ be given. Set $r = (\alpha\beta) \bmod q$, $s = ru_2^{-1}$, $u_1 = H(m)s^{-1}$, and $c = r^{-1}H(m)$. The verification procedure verifies a non-interactive proof that $\log_g \alpha = c \log_h \beta$. This proof is sound assuming H' is a random function, so assume this identity holds. This, together with the identity $u_1 = cu_2$, implies that $g^{u_1} = \alpha$, proving that (r, s) is a valid signature.

Secrecy. Suppose that an adversary can win Game B, obtaining a signature on a message m chosen in B3. Under the strong security assumption for DSS, this signature must be the same as that generated in B3, which implies the adversary can compute $u_2 = \log_h \beta$.

We show how to use this adversary to compute discrete logarithms efficiently, assuming H' is a random function. Let ω be a random element of order q in $\mathbf{Z}_p^*$ whose discrete logarithm to the base g we wish to compute. We choose $x \in \mathbf{Z}_q$ at random, and compute $h = g^x$, and use h as a public key for the signature scheme. Since we know x, we can generate signatures for the adversary as necessary in B2 and B4. Now in B3, we are given a message m by the adversary. We do not sign this message; instead, we compute $r = \omega \bmod q$, and set $c = r^{-1}H(m)$. Then we compute $\beta = \omega^{x/(x+c)}$ and $\alpha = \omega/\beta$. It is easily verified that α and β have precisely the same distribution as in the actual reduction algorithm, so this part of the simulation is perfect. Also, since H' is a random function, we can easily simulate the proof that $\log_g \alpha = c \log_h \beta$, since the corresponding interactive proof is zero knowledge against an honest verifier. Thus, we have perfectly simulated the output of the reduction algorithm. Now if the adversary computes $u_2 = \log_h \beta$, then we obtain $\log_g \omega = (x + c)u_2$.

That completes the proof of Lemma 1.

3.4 Fiat-Shamir Signatures

In the Fiat-Shamir signature scheme, there are two security parameters k and t, and a hash function H that outputs a $t \times k$ bit-matrix. To generate a key, one chooses a composite modulus M, and for $1 \le j \le k$, selects $s_j \in \mathbf{Z}_M^*$ at random, and computes $v_j = s_j^2$. The public key consists of M and $v_1, \ldots, v_k$, and the secret key consists of $s_1, \ldots, s_k$. To sign a message m, for $1 \le i \le t$ the signer chooses $r_i \in \mathbf{Z}_M^*$ at random, computes $w_i = r_i^2$, and then computes $c = H(m, w_1, \ldots, w_t)$. Next, for $i = 1, \ldots, t$, the signer computes $z_i = r_i / \prod_{j=1}^k s_j^{c_{ij}}$. The signature is $(c, z_1, \ldots, z_t)$. To verify a signature, one checks that $H(m, w_1, \ldots, w_t) = c$, where $w_i = z_i^2 \prod_{j=1}^k v_j^{c_{ij}}$.

The Fiat-Shamir scheme is provably secure in the random hash function model (if factoring is hard).

Our reduction algorithm works as follows. We have a signature $(c, z_1, \ldots, z_t)$ on a message m. We output

$$u_1 = z_1^2;\ u_2 = z_1 z_2, \ldots, u_t = z_1 z_t,\ c.$$

The promised inverse is a square root of u_1.

The verification algorithm checks that

$$H(m, u_1 \prod_{j=1}^k v_j^{c_{1j}}, u_2^2/u_1 \prod_{j=1}^k v_j^{c_{2j}}, \ldots, u_t^2/u_1 \prod_{j=1}^k v_j^{c_{tj}}) = c.$$

The recovery algorithm takes a square root z_1 of u_1, and computes $z_i = u_i/z_1$ for $2 \le i \le t$, and outputs the signature $(c, z_1, \ldots, z_y)$.

To prove that this scheme achieves secrecy, we have to make the following *strong security assumption* about the Fiat-Shamir scheme: given signatures on several messages, not only is it difficult to compute a signature on a new message, it is difficult to to compute a signature on any of the given messages with a different $(w_1, \ldots, w_t)$. This strong security assumption holds in the random hash function model if factoring is hard.

Lemma 2 *Under the strong security assumption for the Fiat-Shamir scheme, the above reduction scheme is secure.*

Completeness and soundness are trivial.

To prove secrecy, suppose that an adversary can win Game B. Let m be the message chosen in B3 by the adversary, and let $u_1, \cdots, u_t, c$ be the output of the reduction algorithm. Under the strong security assumption for Fiat-Shamir signatures, the only signature that the adversary can compute is one of the form $(c, z_1', \ldots, z_t')$, where z_1' is a square root of u_1. We show how to use this adversary to factor M.

We start playing Game B against the adversary by generating a public key/secret key pair for the signature scheme. Since we know the secret key, we can generate signatures on demand. It is straightforward to verify that the reduction algorithm is *witness hiding*, in the sense that the z_1 used by the reduction algorithm is distributed uniformly among the square roots of u_1, independently of the output of the reduction algorithm (a similar independence condition holds for the other z_i, but *not* for their joint distribution). Therefore, z_1 and z_1' are independent square roots of u_1, and so with probability 1/2, $\gcd(M, z_1 - z_1')$ is a proper factor of M.

That completes the proof of 2.

3.5 Ong-Schnorr Signatures

Ong-Schnorr signatures are a generalization of Fiat-Shamir signatures with a third security parameter l. The hash function H outputs a $t \times k$ matrix, with each entry in $\{0, \ldots, 2^l - 1\}$. To generate a key, one chooses a composite modulus M, and for $1 \leq j \leq k$, selects $s_j \in \mathbf{Z}_M^*$ at random, and computes $v_j = s_j^{2^l}$. The public key consists of M and $v_1, \ldots, v_k$, and the secret key consists of $s_1, \ldots, s_k$. To sign a message m, for $1 \leq i \leq t$ the signer chooses $r_i \in \mathbf{Z}_M^*$ at random, computes $w_i = r_i^{2^l}$, and then computes $c = H(m, w_1, \ldots, w_t)$. Next, for $i = 1, \ldots, t$, the signer computes $z_i = r_i / \prod_{j=1}^k s_j^{c_{ij}}$. The signature is $(c, z_1, \ldots, z_t)$. To verify a signature, one checks that $H(m, w_1, \ldots, w_t) = c$, where $w_i = z_i^{2^l} \prod_{j=1}^k v_j^{c_{ij}}$.

The Ong-Schnorr signature scheme is provably secure in the random hash function model (if factoring is hard). This was proven for so-called "Blum integers" (where M is a product of two primes equal to 1 modulo 4) by Jakobsson [14], and later extended to more general moduli by Schnorr [22, 23].

Our reduction algorithm works as follows. We have a signature $(c, z_1, \ldots, z_t)$ on a message m. We output

$$u_1 = z_1^2;\ u_2 = z_1 z_2, \ldots, u_t = z_1 z_t,\ c.$$

The promised inverse is a square root of u_1.

The verification algorithm checks that

$$H(m, u_1^{2^{l-1}} \prod_{j=1}^k v_j^{c_{1j}}, (u_2^2/u_1)^{2^{l-1}} \prod_{j=1}^k v_j^{c_{2j}}, \ldots, (u_t^2/u_1)^{2^{l-1}} \prod_{j=1}^k v_j^{c_{tj}}) = c.$$

The recovery algorithm takes a square root z_1 of u_1, and computes $z_i = u_i / z_1$ for $2 \leq i \leq t$, and outputs the signature $(c, z_1, \ldots, z_y)$.

We have to make the same strong security assumption that we made for the Fiat-Shamir scheme (which also is provable in the random hash function model).

Lemma 3 *Under the strong security assumption for the Ong-Schnorr scheme, the above reduction scheme is secure.*

The proof is essentially the same as that for Lemma 2, and we omit the details.

3.6 GQ Signatures

GQ signatures are essentially isomorphic to Schnorr signatures, with the eth map modulo a composite in GQ playing the role of exponentiation in Schnorr. Here, e should be a large prime number. Because of this isomorphism, the reduction from GQ signatures to eth roots is isomorphic to the reduction from Schnorr signatures to discrete logarithms. We leave the details to the reader.

3.7 Anonymous Off-Line Electronic Coins

Various schemes have been proposed for anonymous, off-line cash. For concreteness, we consider a scheme of Brands [5]. In this scheme, there is a group G of order q, and two generators $g_1, g_2 \in G$. A coin consists of two group elements $A, B \in G$, along with a special kind of signature on the coin from the bank. To spend a coin at a given shop, a user generates a payment transcript consisting of $r_1, r_2 \in \mathbf{Z}_q$ such that $g_1^{r_1} g_2^{r_2} = A^d B$, where d is a hash of the coin, the shop's identity, the current data and time, and possibly other details of the transaction.

Let $\theta : \mathbf{Z}_q \times \mathbf{Z}_q \to G$ be the group homomorphism sending (x_1, x_2) to $g_1^{x_1} g_2^{x_2}$. Then reducing a promise of a payment transcript to a promise of a homomorphic inverse is trivial: the promised inverse is a θ-inverse of $A^d B$.

4 Verifiable Encryption of Homomorphic Inverses

Suppose we a have a surjective group homomorphism $\theta : G_1 \to G_2$. We have a publicly known group element $d \in G_2$ and a secret $s \in \theta^{-1}(d)$. We want to encrypt s under the public key of a third party in such a way that it can be publicly verified that when decrypted, an inverse of d is obtained. However, we want to ensure that this verification procedure itself does not reveal any information that helps invert d. We also immutably bind to the encryption an *label* $x \in \{0,1\}^*$, which will be used by the third party to determine if this decryption is authorized.

More formally, a verifiable encryption scheme consists of a key generation algorithm, a prover P, a verifier V, a decryption algorithm D, and a recovery algorithm R. P and V have as a common inputs $d \in G_2$ and $x \in \{0,1\}^*$, along with the public encryption key. P also has a private input $s \in \theta^{-1}(d)$. At the end of the protocol, V either accepts and outputs a string α, or rejects. The string α is a ciphertext that can be given to D, along with the label x; the output from D can be given to R to obtain s.

A *secure* verifiable encryption scheme satisfies the following properties:

Completeness. If both P and V are honest, then for all s, d, and x, with $\theta(s) = d$, V accepts.

Soundness. For all d and x, and for arbitrary P^*, if V accepts and outputs α, then with overwhelming probability, $\theta(R(D(\alpha, x))) = d$.

Zero Knowledge. Consider the following game played against an adversary:

Game C

C1 The key generation algorithm is run, the private key is given to D and the public key is given to the adversary.

C2 The adversary makes arbitrary queries $D(\alpha', x')$.

C3 The adversary generates s, d, x with $\theta(s) = d$, and gives P the input s, d, x, along with the public encryption key.

C4 The adversary makes arbitrary queries to D and P, but after its first query to P, it may not query D with label x.

A simulator is a machine that plays the roles of the key generation algorithm, the decryption oracle D, and the prover P, but is only given d and x—and not s. In the random hash function model, the simulator also responds to the random function queries. *Zero knowledge* means there exists a simulator such that the adversary cannot feasibly distinguish between the real game and the simulated game.

Note that in the definition of zero knowledge, the power of the simulator is quite limited: it must respond to queries on-line, and is not allowed to do anything like "rewind" the adversary.

We now give our verifiable encryption scheme, which is already optimized somewhat to reduce the amount of data transmitted.

First, we assume that we have a public-key encryption function E secure against adaptive chosen ciphertext attack [19]. More precisely, the public key defines a function $E(t, y)$; to encrypt a string y, one chooses a random string t, of length, say, k, and computes $E(t, y)$.

Second, we assume we have hash functions H_1, H_2, and H_3. H_1 takes a string r, of length, say l, and outputs a pair $(t, s') \in \{0,1\}^k \times G_1$. H_2 is a hash function that maps a pair $(\alpha, d') \in \mathsf{Range}(E) \times G_2$ to a short string. H_3 just hashes bit strings to shorter bit strings (but long enough to resist collisions). Also, a security parameter N is defined.

The protocol is a simple "cut and choose" scheme (but with exponential—not linear—security), and runs as follows.

Protocol D

The following steps are executed N times *in parallel.*

D1 P chooses $r \in \{0,1\}^l$ at random, computes $(t, s') = H_1(r)$, and sends $h = H_2(E(t, (s', H_3(x))), \theta(s'))$ to V.

D2 V chooses $b \in \{0, 1\}$ at random, and sends b to P.

D3 If $b = 0$, P sends r to V. If $b = 1$, P sends $\alpha = E(t, (s', H_3(x)))$ and $s'' = s' + s$ to V.

D4 If $b = 0$, V computes $(t, s') = H_1(r)$ and checks that $h = H_2(E(t, (s', H_3(x))), \theta(s'))$. If $b = 1$, V checks that $h = H_2(\alpha, \theta(s'') - d)$. If these checks fail, V rejects.

V rejects if it rejects in any of the N rounds. Otherwise, it accepts and outputs the set of all of the ciphertexts α in those rounds with $b = 1$. Actually, to ensure completeness, V should pick a random string of N bits, not all zero.

Decryption is straightforward: when D is given an label x and a set of ciphertexts $\{\alpha\}$, each ciphertext is decrypted, and the first component of the cleartext is returned if the second component of the cleartext matches $H_3(x)$. The recovery algorithm takes the given values s', and computes $s = s'' - s'$ for the corresponding s'' in protocol D, and outputs s if $\theta(s) = d$ for one of these values.

Lemma 4 *In the random hash function model for H_1, H_2, and H_3 the above scheme is a secure verifiable encryption scheme.*

Completeness. Clear.

Soundness. This is a standard argument, relying only on the presumption that it is impossible to find collisions in H_2.

Zero Knowledge. We now describe the simulator P' that responds to the queries made to P and the random hash functions. At any stage in Game C, responses to the hash function queries are simply made in a random, but consistent, fashion. In stage C3, the simulator receives inputs d, x. In stage C4, the simulator responds to queries made to P as follows.

In step D1, P' just outputs N random strings, representing N outputs of H_2, whose inputs are yet to be determined. In step D3, P' does the following in each of the N rounds, depending on the value of b given in D2.

If $b = 0$, P' picks $r \in \{0,1\}^l$ at random, and sets $(t, s') = H_1(r)$. Then it sets $\alpha = E(t, (s', H_3(x)))$. Now P' has to "backpatch" H_2, making the output of H_2 at the point $(\alpha, \theta(s'))$ equal to the corresponding string output in step D1. It is easily verified that this can almost certainly be done consistently (and if not, we simply quit).

If $b = 1$, P' chooses $s'' \in G_1$ at random, and creates a "dummy" ciphertext α under E of $(0, H_3(x))$. Again, we have to backpatch H_2, this time at the point $(\alpha, \theta(s'') - d)$, and again it is easy to see that this backpatching can almost certainly be done consistently.

That completes the description of the simulator. We next claim that if the adversary can distinguish a simulated Game C from a real Game C, we can use this adversary to break the underlying encryption scheme. To see this, note that in the simulated game, if the adversary ever presents one of the dummy ciphertexts for decryption in stage C4, with label $x' \neq x$, then we do not have to give the underlying decryption algorithm for E the ciphertext, since the decryption algorithm D would almost certainly refuse to give the adversary the cleartext anyway. Then, using a standard argument, the adversary could be used to distinguish an encryption of $(0, H_3(x))$ from an encryption of $(s', H_3(x))$, thus breaking the underlying encryption scheme.

That completes the proof of Lemma 4.

4.1 An Example Implementation: Discrete Logarithms

Assume that H_2 and H_3 have 160-bits of output, and that the input length l of H_1 is 160 bits. Note that the probability that an honest verifier is "cheated" in any one interaction is roughly 2^{-N}; therefore, $N = 40$ should be sufficient for most applications. One could make the protocol non-interactive using standard techniques involving hash functions, but then much larger values of N, say $N = 80$, would be required to avoid off-line attacks.

For our encryption function, we take the OAE encryption function of Bellare and Rogaway [4], based on the RSA problem. Assume a composite modulus of 1024 bits and an encryption exponent of 3. OAE is secure against chosen ciphertext attacks in the random hash function model (although the proofs in [4] have to be adapted slightly to prove this).

Apropos DSS, for the discrete logarithm problem, assume the group G_2 is the subgroup of order q in $\mathbf{Z}_p^*$, where p is a 1024-bit prime, and q is a 160-bit prime. Let g be the given generator for G_2. In our notation, the group G_1 is the additive group $\mathbf{Z}_q$, and θ sends $a \in \mathbf{Z}_q$ to g^a.

The expected amount of data transmitted is about 4 KBytes.

Both P and V perform 40 160-bit exponentiations in $\mathbf{Z}_p^*$, all to the base g, plus the multiplications for OAE. Using techniques of Lim and Lee [16], each party can do this using under 2000 modular multiplications. This bound already includes the precomputation time for the base g. If g is actually fixed for one of the parties, this bound can be reduced to about 1000 modular multiplications.

4.2 An Example Implementation: RSA Inverses

Assume the same encryption function as above.

For the RSA inverse problem, assume a 1024-bit composite modulus M and encryption exponent e with $(\phi(M), e) = 1$. In our notation, $G_1 = G_2 = \mathbf{Z}_M^*$, and $\theta(a) = a^e$. Typically, $e = 3$ or

$e = 2^{16} + 1$. One could also take $e = 2$, in which case $G_2 = (\mathbf{Z}_m^*)^2$.

The expected amount of data transmitted is about 7 KBytes.

For $e = 3$, each party needs no more than 160 modular multiplications, and for $e = 2^{16} + 1$, this number is under 800.

5 The Fair Exchange Protocol

Now that we have all the necessary tools, we can easily describe our fair exchange protocol.

Suppose A holds a signature σ_A on message m_A under public key PK_A, and B holds a signature σ_B on message m_B under public key PK_B

We make use of our scheme for reducing signatures to homomorphic inverses. Let θ_A be the relevant homomorphism for A's signature, and θ_B be that for B's signature. We write $\mathsf{desc}(\theta_B)$ for a string that describes the relevant groups and an algorithm for computing the homomorphism using some standard encoding.

We also make use of our scheme for verifiable encryption of homomorphic inverses under T's public key. Recall that when T decrypts such a verified encryption, it always does this subject to a label bound to the encryption at the time the encryption was created.

We also make use of a one-way function f.

The third party T maintains a set S of tuples, whose structure is described below. We describe the protocol assuming A makes the first move.

Protocol E

E1 A computes $\mathsf{reduce}(PK_A, m_A, \sigma_A) = (d_A, c_A, s_A)$, and sends d_A, c_A to B.

E2 B checks that $\mathsf{verify}(PK_A, m_A, d_A, c_A)$ accepts; if not, B halts. Otherwise, B computes $\mathsf{reduce}(PK_B, m_B, \sigma_B) = (d_B, c_B, s_B)$, and sends d_B, c_B to A.

E3 A checks that $\mathsf{verify}(PK_B, m_B, d_B, c_B)$ accepts; if not, A halts. Otherwise, A chooses $r \in \mathsf{Domain}(f)$ at random and computes $v = f(r)$. A and B then engage in the verifiable encryption protocol, with A as prover and B as verifier, so that A gives to be B a verified encryption α of s_A with label $(v, d_B, \mathsf{desc}(\theta_B))$.

E4 If B rejects the proof in E3, then B halts. Otherwise, B and A engage in the verifiable encryption protocol, with B as prover and A as verifier, so that B gives to A a verified encryption β of s_B with label v.

E5 If A rejects the proof in E4, then A invokes sub-protocol abort. Otherwise, A sends s_A to B.

E6 B checks that $\theta_A(s_A) = d_A$. If not, B invokes sub-protocol $\mathsf{B\text{-}resolve}$. Otherwise, B sends s_B to A, outputs $\mathsf{recover}(PK_A, m_A, c_A, s_A)$, and halts.

E7 A checks that $\theta_B(s_B) = d_B$. If not, A invokes sub-protocol $\mathsf{A\text{-}resolve}$. Otherwise, A outputs $\mathsf{recover}(PK_B, m_B, c_B, s_B)$, and halts.

Sub-protocol abort

A sends $r, d_B, \mathsf{desc}(\theta_B)$ to T, who does the following:

if $(\mathsf{deposit}, f(r), d_B, s_B, \mathsf{desc}(\theta_B)) \in S$ then
  send s_B to A, from which A recovers σ_B
else if $(\mathsf{no\text{-}abort}, f(r)) \in S$ then
  send A the message "abort not allowed"
else
  add $(\mathsf{abort}, f(r))$ to S
  send A the message "exchange aborted"

Sub-protocol B-resolve

B sends $\alpha, v, s_B, \mathsf{desc}(\theta_B)$ to T, who does the following:

if $(\mathsf{abort}, v) \in S$ then
  send B the message "exchange aborted"
else
  add $(\mathsf{deposit}, v, \theta_B(d_B), s_B, \mathsf{desc}(\theta_B))$ to S;
  decrypt α subject to the label $(v, \theta_B(s_B), \mathsf{desc}(\theta_B))$,
  and send result to B, from which B recovers σ_A.

Sub-protocol A-resolve

A sends A-resolve$, \beta, r$ to T, who does the following:

if $(\mathsf{abort}, f(r)) \in S$
  send A the message "exchange aborted"
else
  add $(\mathsf{no\text{-}abort}, f(r))$ to S;
  decrypt β subject to the label $f(r)$,
  and send result to A, from which A recovers σ_B

Our main result is the following.

Theorem 1 *Assuming that f is one-way, that the underlying reduction and verifiable encryption schemes are secure (in the random hash function model), then the above fair exchange protocol is secure (in the random hash function model).*

Remarks. (1) Using standard techniques, s_A and s_B can be blinded, so that if both A and B are honest, T obtains no useful information. (2) In principle, when a tuple is added to S, it must stay there forever; in practice, an "aging" mechanism can be introduced, allowing old tuples to be eventually flushed.

The intuition for the proof of fairness is that at no point does the adversary obtain any useful information about the honest player's signature before it becomes essentially inevitable that the honest player will obtain the adversary's signature.

We now prove Theorem 1.

Completeness is clear. To prove fairness, we start with an adversary that can win Game A. Consider first the case where A is the honest player, and call the adversary B^*.

We begin by defining a truncated version of Game A, call it Game A$'$, defined by the following two early-stopping rules:

(1) Suppose that E4 has just completed with A accepting the proof in verifiable encryption protocol. Then we stop the game at this point, and say that B^* loses.

(2) Suppose that A has begun to engage in the encryption protocol in E3 with a given label, and A has not executed the abort sub-protocol. If these conditions hold, and B^* sends a request to T for B-resolve with arguments that yield a matching label, we stop the game just before T responds, and say B^* loses.

We claim that if B^* can win Game A, then he can also win Game A$'$. To see why, consider stopping rule (1). Once A accepts the verifiable encryption, it is effectively inevitable that A would obtain s_B, and hence σ_B, in Game A: either directly in step E6, or via sub-protocol A-resolve. In the latter case, since f is one-way, the adversary cannot block the A-resolve by having an abort tuple placed in S via sub-protocol abort.

Now consider stopping rule (2). Since A has not aborted the transaction (and only A can do this as f is one-way), then performing the B-resolve in Game A would result in a deposit tuple being placed on S that contains an inverse of s_B. Then A will eventually get s_B, and hence σ_B, either from B^* directly in step E6, or from T via abort or A-resolve, which again B^* cannot prevent since it cannot invert v.

So assume B^* can win Game A$'$. We now make another transformation: we replace the A's prover in the verifiable encryption protocol in E3 (and any random hash functions) with the corresponding zero-knowledge simulator. Because of the way we truncated Game A, we are guaranteed that T's decryption function will never be called with a matching label at any time after the prover starts.

Call this Game A$''$. By the zero-knowledge property, it follows that B^* can also win Game A$''$. Moreover, when we play Game A$''$ against the adversary, we do not need the s_A output from the reduction algorithm to run the prover simulation, and it is not in the adversary's view, so we can drop it altogether. The result is an adversary that breaks the supposed secrecy of A's reduction scheme.

That concludes the proof for the case where A is the honest player. Now assume B is the honest player, and call the adversary A^*. We use a very similar argument.

We first define a truncated Game A$'$ via three early-stopping rules:

(1) If A^* sends B a valid inverse in E5, we stop, and say that A^* loses.

(2) If B has completed step E3, accepting the proof in the verifiable encryption, and A^* has not performed a successful abort with v, and then A^* performs an A-resolve with v, then we stop the game just before T responds, and say that A^* loses.

(3) If B has just accepted the proof E3, but A^* has already performed a successful A-resolve with v, then we stop, and say that A^* loses.

As before, we show that if A^* can win Game A, it can also win Game A$'$. Clearly, if we stop on rule (1), then in Game A, B would recover σ_A, and so A^* would lose. Also, if we stop on rules (2) or (3), then in Game A, T would add a no-abort tuple that would prevent A^* from aborting the transaction at any point in the future, thus allowing B to obtain s_A, and hence σ_A, via B-resolve should it need to.

The rest of the proof goes exactly the same, *mutatis mutandis*, as in the case above where A was honest.

6 Verifiable Encryption with Off-Line Coupons

In this section, we describe a more efficient solution to the verifiable encryption problem. Since we are already using a trusted third party for the fair exchange protocol, we can use a trusted third party to generate certified "coupons" that can be used to efficiently generate and validate the encryptions that we need. These coupons can be obtained in bulk and off line, before performing any exchanges.

As above, assume we have a group isomorphism $\theta : G_1 \to G_2$. We assume that the trusted third party T has a public signing key.

A *coupon* is a triple of the form (d', α, PK), where $d' \in G_1$, $\alpha \in \mathsf{Range}(E)$, and PK is a public key for some secure signature scheme.

To obtain a coupon, a player sends a request for a coupon to T. Upon receipt of this request, T generates $s' \in G_1$ and $t \in \{0,1\}^k$ at random, and also generates a secret key/public key pair (SK, PK) for the signature scheme. T then forms the coupon $(\theta(s'), E(t, s'), PK)$, signs it, and sends the coupon, the signature on the coupon, along with s' and SK to the player.

To send a verifiable encryption of (s, x), where $x \in \{0,1\}^*$ is a label and $d = \theta(s)$, to another player, the sender sends the above coupon (d', α, PK), T's signature on the coupon, $s'' = s + s' \in G_1$, and a signature on (α, x) using the key SK.

Upon receipt of these items, the receiver checks that $\theta(s'') = d + d'$, verifies T's signature on the coupon using T's public key, and verifies the signature on (α, x) using the key PK. Clearly, if the receiver obtains a decryption of α, he can readily compute $s = s'' - s'$.

In our fair exchange protocol, protocols A-resolve and B-resolve will have to be modified as follows. When a player sends a resolve request to T, he must send a coupon (d', α, PK) to T, along with x, T's signature on the coupon, the signature on (α, x), as well as the item described by x. Besides the other checks that T performs, T also checks the validity of both of the signatures presented to it.

It is easy to verify that the fair exchange protocol remains secure using this implementation of verifiable encryption, provided coupons are never used more than once.

There are several variants which one might consider. First, a player might generate (SK, PK) and s' on his own, instead of having T do this. Second, the public key encryption function E might be replaced with a symmetric-key cipher, whose key is known to T. Or alternatively, T could just store s' and later reveal it, but this is perhaps impractical.

Note that in our construction, a coupon can be obtained by a player in advance of knowing the type of item it expects to receive in exchange for the item it is giving. Construction of the coupon does depend, however, on the type of item the player is giving. But in many practical situations, this is quite acceptable. For example, a player who wishes to purchase items using his electronic checks can obtain many coupons in advance corresponding to the signature scheme used for his electronic checks. A travel agent selling airline tickets can obtain coupons corresponding to the signature scheme used to sign electronic airline tickets.

7 An Accountable Protocol for Contract Signing

In this section we consider the contract-signing problem: two players wish to sign a contract m in such a way that either each player obtains the other's signature, or neither player does.

We could, of course, use the protocols previously described in this paper. However, if we allow ourselves the flexibility to prescribe the form of the signatures, we can obtain a protocol that is much more efficient, and which achieves another important property lacking in the previous protocols: accountability.

By *accountability*, we mean that if the trusted third party T misbehaves, then this can be *proven*. Suppose that after executing the protocol, A does not obtain a valid contract. If T does not misbehave, then B will not have a valid contract either. However, if B *does* have a valid contract, and ever tries to enforce it in court, then the contract that B presents in court, together with information obtained by A during the execution of the protocol, can be used to *prove* that T misbehaved. At this point, the contract is ruled invalid, and both A and B sue T for misbehaving. Of course, it must be infeasible for A and B to collude to somehow *frame* T if T does not misbehave.

Note that we can combine such an accountable contract-signing protocol with our general exchange protocols. If A wants send B an electronic check in exchange for an electronic airline ticket, then in stage 1, A and B sign a contract to this effect using an accountable contract-signing protocol. In stage 2 they use our more general (but unaccountable) fair exchange protocol to actually exchange the items. If something goes wrong, both parties still have legal recourse: either the contract can be enforced via legal action, or T can be sued for misbehaving in stage 1. Note that even with the contract from stage 1, it is still worthwhile to use a fair exchange protocol in stage 2, as one generally wants to avoid the time and expense of going to court to enforce a contract.

7.1 The Protocol

We now describe our contract-signing protocol. For $X \in \{A, B, T\}$, we denote by $[\alpha]_X$ the string α, concatenated with a signature of α under X's public key. T maintains a set S of messages.

protocol F

F1 A sends $\alpha = [m, A, B, T]_A$ to B.

F2 If B does not receive a valid signature α, it simply quits the protocol. Otherwise, B sends $\beta = [m, A, B, T]_B$ to A.

F3 If A does not receive a valid signature β, it invokes protocol abort. Otherwise, A sends $\alpha' = [m, A, B]_A$ to B.

F4 If B does not receive a valid signature α', it invokes protocol resolve. Otherwise, B sends $\beta' = [m, A, B]_B$ to A.

F5 If A does not receive a valid signature β', then A invokes protocol resolve.

We summarize the message flows:

$A \to B: \ [m, A, B, T]_A$

$B \to A: \ [m, A, B, T]_B$

$A \to B: \ [m, A, B]_A$

$B \rightarrow A: \ [m, A, B]_B$

Sub-protocol abort

A sends to T the message $[m, A, B, \mathsf{abort}]_A$. T verifies the signature, and then checks if their is a message of the form $[[m, A, B, T]_A, [m, A, B, T]_B]_T$ in S. If so, T returns this message to A. Otherwise, T returns $[[m, A, B, \mathsf{abort}]_A]_T$ to A, and also stores this message in S.

Sub-protocol resolve

A player sends to T the message $([m, A, B, T]_A, [m, A, B, T]_B)$. T verifies the signatures, and checks if there is a message of the form $[[m, A, B, \mathsf{abort}]_A]_T$ in S. If so, T returns this message to the player. Otherwise, T generates $[[m, A, B, T]_A, [m, A, B, T]_B]_T$, sends this message to the player and stores it in S.

A *valid contract* on m between A and B is of the form $([m, A, B]_A, [m, A, B]_B)$ or $[[m, A, B, T]_A, [m, A, B, T]_B]_T$, or if it is the above form with the roles of A and B reversed.

If ever a pair strings of the form $[[m, A, B, \mathsf{abort}]_A]_T$ and $[[m, A, B, T]_A, [m, A, B, T]_B]_T$ are produced, T will be judged to have *cheated.*

It is assumed that whenever T is given a query of the form $([m, A, B, T]_A, [m, A, B, T]_B)$ or $[m, A, B, \mathsf{abort}]_A$, then T can be *compelled* to give a response of the form $[[m, A, B, T]_A, [m, A, B, T]_B]_T$ or $[[m, A, B, \mathsf{abort}]_A]_T$. Such an assumption is reasonable, as T is supposed to be a well-known entity providing a guaranteed service; if T is not able to respond within a reasonable amount of time with a valid response, it could be held liable for failure to provide service.

7.2 Security analysis

Security for an honest player means the following. Suppose an adversary has corrupted the other player and T. Then at the end of the protocol, if the adversary obtains a valid contract, but the honest player does not, then the adversary's valid contract, together with information obtained by the honest player, can be used to prove that T cheated.

Security for T means that if T is honest, an adversary cannot produce evidence that T cheated (thereby framing T).

Security for A. Suppose A is honest and does not obtain a valid contract at the end of the protocol.

Suppose that A terminated the protocol via the abort sub-protocol. Then A must have a string of the form $[[m, A, B, \mathsf{abort}]_A]_T$. If the adversary has a valid contract, it must be of the form $[[m, A, B, T]_A, [m, A, B, T]_B]_T$. These two items can thus be used to prove that T cheated.

A could not have terminated using sub-protocol resolve, since A never generated $[m, A, B, \mathsf{abort}]_A]$, and hence the only possible response from T in this case is a valid contract $[[m, A, B, T]_A, [m, A, B, T]_B]_T$.

Security for B. Suppose that B is honest and does not obtain a valid contract.

If B had simply quit the protocol in F2, the adversary can not possibly have a valid contract, as B never signed anything.

If B terminated via sub-protocol resolve, then B must have a string of the form $[[m, A, B, \mathsf{abort}]_A]_T$. If the adversary has a valid contract, it must be of the form $[[m, A, B, T]_A, [m, A, B, T]_B]_T$. These two items can be used to prove that T cheated.

Security for T. It is clear from the definitions of protocols abort and resolve that an uncorrupted T will never output both $[[m, A, B, T]_A, [m, A, B, T]_B]_T$ and $[[m, A, B, \mathsf{abort}]_A]_T$.

8 Conclusion

We close by discussing two directions for future research related to this topic.

First, it would be nice to find a more efficient method of performing verifiable encryption, and in particular avoid the "cut and choose" method that we use. Even a less general or less secure, but more efficient, method would be of interest.

Second, the fact that our trusted third party is off-line means that we could afford to implement it as a distributed, fault-tolerant system, making it highly secure using potentially expensive cryptographic techniques, and eliminating the single point of failure. In future work, we intend to design and implement such a system. To achieve efficiency and provable security (in a random hash function model), the basic protocols require some modification. In particular, a threshold cryptosystem secure against chosen ciphertext attack is required. There has been very little work done on this problem, but recent progress has been made in [25]. Also, it is apparently necessary to solve a sort of Byzantine agreement problem among the distributed servers, who must decide whether a particular transaction is to be "aborted" or "resolved."

References

[1] N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for fair exchange. In *4th ACM Conference on Computer and Communication Security*, pages 6–17, 1997.

[2] M. Bellare and S. Goldwasser. Encapsulated key escrow. Preprint, 1996.

[3] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *First ACM Conference on Computer and Communications Security*, 1993.

[4] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology—Crypto '94*, pages 92–111, 1994.

[5] S. Brands. Untraceable off-line cash in wallets with observers. In *Advances in Cryptology–Crypto '93*, pages 302–318, 1993.

[6] H. Bürk and A. Pfitzmann. Value exchange systems enabling security and unobservability. *Computers and Security*, 9:715–721, 1990.

[7] D. Chaum and T. Pederson. Wallet databases with observers. In *Advances in Cryptology–Crypto '92*, pages 89–105, 1992.

[8] B. Cox, J. D. Tygar, and M. Sirbu. NetBill security and transaction protocol. In *First USENIX Workshop on Electronic Commerce*, pages 77–88, 1995.

[9] R. H. Deng, L. Gong, A. A. Lazar, and W. Wang. Practical protocols for certified electronic mail. *J. of Network and Systems Management*, 4(3), 1996.

[10] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Advances in Cryptology—Crypto '86*, pages 186–194, 1986.

[11] M. K. Franklin and M. K. Reiter. Verifiable signature sharing. In *Advances in Cryptology–Eurocrypt '95*, pages 50–63, 1995.

[12] M. K. Franklin and M. K. Reiter. Fair exchange with a semi-trusted third party. In *4th ACM Conference on Computer and Communications Security*, pages 1–5, 1997.

[13] L. Guillou and J. Quisquater. A "paradoxical" identity-based signature scheme resulting from zero-knowledge. In *Advances in Cryptology–Crypto '88*, pages 216–231, 1988.

[14] M. Jakobsson. Reducing costs in identification protocols. Manuscript available at `http://www-cse.ucsd.edu/users/markus`, 1992.

[15] D. W. Kravitz. Digital signature algorithm, 1993. U. S. Patent No. 5,231,668.

[16] C. H. Lim and P. J. Lee. More flexible exponentiation with precomputation. In *Advances in Cryptology–Crypto '94*, pages 95–107, 1994.

[17] S. Micali. Certified e-mail with invisible post offices. Unpublished manuscript, 1997 (presented at the *1997 RSA Security Conference*).

[18] H. Ong and C. Schnorr. Fast signature generation with a Fiat Shamir-like scheme. In *Advances in Cryptology–Eurocrypt '90*, pages 432–440, 1990.

[19] C. Rackoff and D. Simon. Noninteractive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology–Crypto '91*, pages 433–444, 1991.

[20] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, pages 120–126, 1978.

[21] C. Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4:161–174, 1991.

[22] C. Schnorr. Security of 2^t-root identification and signatures. In *Advances in Cryptology–Crypto '96*, 1996.

[23] C. Schnorr. Erratum: Security of 2^t-root identification and signatures. In *Advances in Cryptology–Crypto '97*, 1997.

[24] V. Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology–Eurocrypt '97*, 1997.

[25] V. Shoup and R. Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. Manuscript, 1997.

[26] M. Stadler. Publicly verifiable secrete sharing. In *Advances in Cryptology–Eurocrypt '96*, pages 190–199, 1996.