How to Maintain Authenticated Communication in the Presence of Break-ins

(Extended Abstract)

Ran Canetti * Shai Halevi † Amir Herzberg‡

May 8, 1996

Abstract

Cryptography provides authenticated communication over untrusted channels, as long as the secret keys are not exposed. However, attacks by hackers and insiders often expose secret keys. Such break-in attacks often control the systems only for a limited time, and therefore security may be regained, provided new keys can be selected and installed securely. This proactive recovery operation must be invoked periodically, since the exposure may remain undetected.

We consider a highly adversarial scenario where the adversary has complete control over the network, and can also occasionally break into parties, expose and modify their keys. In this setting we present a mechanism which manages to maintain authenticated communication. Using our scheme as a stepping stone, general "higher level" applications for secure multiparty computation, distributed databases, secret sharing , etc., can be performed in this scenario.

Keywords: Authentication protocols, break ins, recovery, distributed signatures, proactive protocols.

*LCS - MIT, NE43-336, 545 Tech. Square, Cambridge MA 02139, USA. e-mail: canetti@theory.lcs.mit.edu. Part of this research was done while visiting IBM T.J. Watson Research Center.

†LCS - MIT, NE43-344, 545 Tech. Square, Cambridge MA 02139, USA. e-mail: shaih@theory.lcs.mit.edu.

‡IBM T.J. Watson Research Center, POB 704, Yorktown Heights, NY 10598, e-mail: amir@watson.ibm.com

1 Introduction

Maintaining authenticated communication over an untrusted network is perhaps the most basic goal of cryptography. Practically no cryptographic application can get off the ground without authenticity. Today, with the extensive use and wide availability of electronic communication this goal is ever so relevant.

Authenticated communication can be maintained using a variety of standard cryptographic techniques, even if the communication links are fully controlled by an adversary. However, all cryptographic techniques rely on the ability of the communicating parties to maintain the integrity and secrecy of some data, namely their cryptographic keys. Indeed, attacks on the security of the communication systems are often based on obtaining, or even modifying, these keys. We call such attacks **break-in attacks**. How can authenticated communication be maintained in the presence of repeated break-in attacks? Answering this question is the focus of our paper.

The contribution of this paper is twofold. On the more fundamental level, we present a rigorous model for the problem of communicating over unauthenticated links with break-ins, sort out what can and cannot be done in this model, and give quantitative measures to the "quality" of a solution. On the practical level, we present a surprisingly effective solution to this problem. Our solution involves a novel use of distributed signature schemes, such as the ones recently suggested in [19, 20, 13].

THE PROBLEM. We consider parties that communicate via a network that is "totally unreliable". This is modeled by assuming that an adversary has full control over the network. That is, the adversary can read, modify, delete and duplicate messages sent over the network, or even inject its own messages. Furthermore, the adversary may occasionally break into parties and obtain all their secret data; even further, this data may be replaced by fake one. Of course, if we are to offer some protocol for solving this problem, we should make sure that the adversary cannot modify the code of this protocol itself. Therefore we assume that the *program* of each party is written in a Read-Only-Memory (ROM) and cannot be modified.

Since we envision a system which is running for a relatively long time, it will be unreasonable to assume any limit on the overall number of parties that can be broken into throughout the lifetime of the system. We only assume, then, that not too many parties are broken into *at the same time*. Following the proactive approach in [18, 10, 15] we introduce periodical, short **refreshment phases**, during which the parties jointly try to refresh their keys and regain security. Let a **time-unit** denote the time between two consecutive refreshment phases. We assume that the number of parties which are broken into during a single time-unit is not too large.

Let us gain more intuition for the problems this model suggests by describing some unavoidable attacks in this model. The first attack is trivial: the adversary may fail to deliver (all, or some) messages. This demonstrates that we can never guarantee that the computation will proceed as if the links were reliable (e.g., in case of a network partition). However, we can expect the parties to be *aware* of the fact that messages are not faithfully delivered. A party that notices malfunction can output a special `alert` signal to a higher layer protocol or an operator.

The second attack is more subtle. Consider a party P that was recently broken into. Then the adversary knows the party's private keys. The adversary, controlling the communication, can now 'cut off' P from the rest of the parties, and impersonate P in their eyes. The impersonation can be kept up even after P is no longer broken into, and *no party other than P can ever notice this attack.* These **impersonation attacks** too are unavoidable in our model; again, the most we can hope for is that the impersonated party will be *aware* of it and output an `alert` signal.

DEFINING OUR GOALS. In light of the above unavoidable attacks, one may wonder if there is any meaningful way to define a "solution" for the problem of authenticated communication in this highly adversarial model. Formulating a reasonable notion of a solution is one of the main contributions of this paper.

We want to devise a general tool that enables protocols to run on an unauthenticated network with break-ins 'as if' it was an authenticated network. In other words, we want to construct an 'emulator' that transforms protocols designed for authenticated networks into ones which run over unauthenticated networks, while maintaining their functionality as much as possible. In Section 2 we define two notions of maintaining functionality, called **awareness** and **simulatability**. Very roughly, we say that an emulator has "awareness" if any protocol running on top of it enjoys the assurance that every deviation from a "perfect execution" will be noticed by some party. Consequently, such an emulator guarantees that as long as no party outputs `alert`, the functionality of the protocol is the same as in an authenticated, reliable network. Simulatability is a stronger notion of preserving functionality. Intuitively, we say that a protocol is "simulatable" if tampering with the links "does not help the adversary": any attack by an adversary that controls the links can also be mounted by a weaker adversary who does not control the links. Consequently, such an emulator guarantees that the functionality of the protocol is *unconditionally* the same as in an authenticated network.

Clearly, neither of these properties can be maintained unless we restrict the power of the adversary (say, by disallowing the adversary to break into all parties at the same time). We measure the quality of an emulator by how powerful an adversary can be tolerated while maintaining simulatability or awareness.

We remark that our problem is related to the problem of entity and session authentication [2, 5, 7, 3]. There authentication was defined as a goal by itself, with emphasis on the ability to run multiple authenticated sessions at the same time. Here we regard authentication as a tool for performing more general multiparty tasks in the presence of unauthenticated communication, and concentrate on the ability to maintain authenticity in the presence of break-ins.

TOWARDS A SOLUTION. Consider a setting where the parties get signature-keys for some signature system at system set-up, along with the verification keys of the other parties. In this setting, authentication is straightforward: each party can sign each of the outgoing messages, and a received message is accepted only if verification of the senders' signature succeeds. (Alternatively, parties can exchange symmetric session keys and use them to authenticate messages). In our model, however, the adversary periodically breaks into parties, exposing (and perhaps modifying) their keys. A first step towards a solution will thus be to let each party replace, at each refreshment phase, its old keys with a freshly chosen pair of signature and verification keys. This way, the new signature key remains secret as long as the party is not broken-into again.

The crux of the problem is how to make sure that other parties learn the new verification key. This is a non-trivial task: consider a party P just recovering from a break-in. P's old signature key is compromised. Thus, the adversary can forge P's signature and send a fake new verification key in the name of P, thus successfully impersonating P. Indeed, as explained above, this kind of attack is unavoidable in our model. However, we want at least to make sure that P is aware of the fact that the adversary is impersonating it. One may attempt to achieve awareness using signed "echos": Let each party Q acknowledge P's new verification key back to P and sign the acknowledgment using Q's signature key, so P can verify that Q got its new public key. Indeed, such a solution may work if P still has Q's verification key. However, the adversary can erase Q's verification key in P's memory, and plant a fake key (for which the adversary knows the corresponding signature key) in its place. The adversary can now impersonate P in the eyes of Q (and, in fact, in the eyes of all other parties) and at the same time impersonate all other parties in the eyes of P!

This brings us to the main technical difficulty in our model. Namely, we need to have a way of "booting trust" during refreshment phases, *when a party cannot even trust its own memory.* We get around this seemingly hopeless situation by letting each party keep a small *fixed* public key in an unmodifiable ROM together with its program, and use this key to bootstrap trust. We outline our solution below.

Our solution. Our main tool is a **proactive, robust, distributed signature scheme (PDS).** Distributed signature schemes can be very roughly described as follows. Each party has its share of a private signature key, in a way that enables any large enough subset of the parties to jointly generate a valid, unforgeable signature of a given message, while preventing the adversary from generating valid signatures. Signatures can be verified using a public verification key. In a *proactive* signature scheme, the parties refresh their shares of the private signature key at each refreshment phase. This is done in a way that prevents the adversary from generating valid signatures as long as not too many parties are broken into between two consecutive refreshment phases. We stress that, even though the shares of the signature key are changed periodically, the public verification key *remains unchanged throughout.*

Each party can now keep a copy of the unchanging, public verification key of a PDS scheme in a small ROM. Thus, each party is always able to verify (and trust) signatures with respect to this key. Such signatures can be generated jointly by the parties.

Using an appropriate PDS scheme, authenticated communication is achieved as follows: At each refreshment phase each party chooses a new set of 'personal' signature and verification keys of some (standard) signature scheme. Next, the party obtains a **certificate**, generated jointly by all parties using the PDS scheme, for the new verification key. (The certificate may read: *"it is certified that the personal verification key of P_i for time unit ℓ is ...".*)[1] Next some more work is done in order to maintain the PDS scheme itself. Each party now signs each of its messages in this time-unit using its current signature key attaches the corresponding verification key, along with the newly obtained certificate. The recipient of a message verifies the certificate using the verification key in his secure ROM, and then verifies the senders' signature of the message.

If a party fails to obtain a certificate for its new verification key, or if it fails to refresh its share of the PDS scheme, then this party issues an `alert` signal which signals that this party is under attack by the adversary and may not be able to authenticate its communication. This alert it handled out-of-band by a higher level protocol or the operator. Our protocol guarantees that this happens only if the party is actually being 'cut-off' from a very large fraction of the network.

We show that this simple-to-implement construction achieves awareness as long as less than half the parties are broken into during each time unit. Simulatability is achieved as long as the adversary does not render more than half of the parties 'non-operational'. (Roughly, a party is 'non-operational' if it is either broken into or 'disconnected' from most of the other parties.) Exact statements can be found in Section 2.

For our construction we can use several known proactive signature schemes, such as the ones described in [19, 20, 13]. We also describe yet another proactive signature scheme, that is simple and specially tailored to our needs. (In fact, our scheme may be of independent interest and useful in other scenarios as well).

Organization. In Section 2 we define the model of computation and the adversary, and present a general definition of a scheme for emulating authenticated network on an unauthenticated one, in the presence of repeated break-ins. In Section 3 we describe tools used in our construction. In Section 4 we describe our construction. Section 5 describes our alternative distributed signature scheme. We sketch a proof of security in the appendix.

[1] To use our protocol in a very large network (e.g., the Internet), we use the standard "hierarchy-approach" for certification. That is, we network is partitioned to local neighborhoods according to some hierarchy, and the protocol is only performed in local neighborhoods (with each neighborhood run its own PDS scheme). A party will obtain a certificate from one or more of the neighborhoods it belongs to.

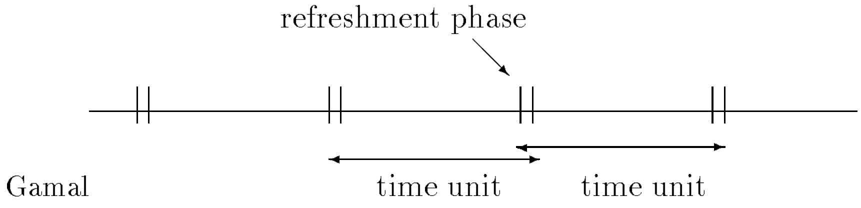

Figure 1: Time units and refreshment phases.

2 Definitions

In this section we define the model of computation and the adversary, and formalize the requirements from a protocol for maintaining authenticated communication in this model. The central concept in our definitions is an **emulator**, which allows any protocol assuming an authenticated network to run over an unauthenticated network. The formal definitions may be skipped without making the construction any less clear.

THE MODEL OF COMPUTATION AND THE ADVERSARY. We consider a synchronous network of n parties, where the communication channels are unauthenticated and controlled by an adversary. This is modeled as follows. All parties have access to a black-box representing the network. (We call this black-box **the network**.) At each communication round, each party P sends its messages by handing them to the network on a special **send** tape, dedicated to P. Next P reads the messages delivered to him in this round on a `receive` tape dedicated to P.

Since we model a network with unauthenticated channels, we assume that there is an *adversary* which controls the network, and can write anything on the parties' receive tapes. We say that a message from party P to party Q was **faithfully delivered** if the text written by P on its send tape, addressed to Q, appears unmodified on Q's receive tape, in the same round. [2]

On top of controlling the network, the adversary may also occasionally **break into** parties during the course of the computation. Once a party is broken into, the entire contents of the party's memory becomes known to the adversary. The adversary can also modify the memory contents of a corrupted party, except for its program. (We think of the program as being written in ROM.) Once the adversary **leaves** a party it no longer can read or modify the party's internal data. (Of course, the adversary remembers data read in the past).

Following the proactive approach, we divide the lifetime of the system into *time-units* (which may be days, weeks, months etc., as appropriate for the specific application). We introduce short **refreshment phases**, that take place at the beginning of each time-unit. The duration of a refreshment phase is typically several communication rounds (i.e., several milliseconds to seconds). This is much shorter than the time-unit itself. Formally we define a time-unit to be the time period that starts at the beginning of a refreshment phase and ends at the end of the succeeding refreshment phase (so each refreshment phase belong to both the previous and next time-unit). See Figure 1. We stress that a time unit may span many communication rounds.

MEASURING THE POWER OF AN ADVERSARY. We define two "measures of power" for the adversary. The first measure is fairly standard: an adversary is said to be t-**break-in-limited** if at most t parties are broken into during any time unit. Note that a t-**break-in-limited** adversary is unlimited in the number of connections it can 'hijack'. In particular, such an adversary can mount the 'impersonation attack' described in the Introduction on all the parties, one by one, thus gain control over the entire system.

[2] For simplicity we assume that each party sends at most one message to each other party at each round.

The second measure is slightly more involved. Here we also limit the adversary's power to "disconnect" parties from 'the rest of the network' by failing to faithfully deliver messages. Roughly, an adversary is (r, s)-disconnection-limited if at each time unit there are at most s 'non-operational' parties, where a 'non-operational' party is one which is broken into, or has less than r 'good' links to 'the rest of the network'. More precisely, we say that a link between two parties P_i and P_j is faithful at a time unit if all the messages sent between P_i and P_j in this time-unit are faithfully delivered. We define r-*operational* parties inductively as follows. Initially (i.e., at time-unit 1) the r-operational parties are those which are not broken into. A party P_i is r-operational at time unit $\ell > 1$ if:
(a) P_i is not broken into during time unit ℓ; and
(b) during time unit ℓ, P_i has faithful links with at least r parties that were r-operational at time unit $\ell - 1$.
An adversary is (r, s)-disconnection-limited if at each time unit at most s parties are not r-operational. Note that an (r, t)-disconnection-limited adversary is also a t-break-in-limited adversary. The converse is not true (unless $r = 0$).

AN AUTHENTICATED NETWORK EMULATOR. The goal of an emulator is to allow application protocols to run over an unauthenticated network as if it was a reliable, authenticated network. Alas, this is not possible since unauthenticated networks can always disrupt the communication by simply omitting messages. We thus settle for limiting the network to only omitting messages. This is captured as follows. We first define an authenticated network as a network restricted to omission failures. Using the above formulation of a network, an authenticated network may write a message on P's receive tape only if this message was sent, addressed to P, in this round.

We consider emulators Λ that operate roughly as follows. Let Π be any protocol that assumes an authenticated network. We run Λ on Π to generate a protocol, denoted $\Lambda\Pi$, that can be thought of as having two layers. At the top layer Π runs unchanged. At the bottom layer, for each message that Π instructs to send, the parties will follow some procedure specified by Λ. Also, in each refreshment phase the parties execute some refreshment protocol specified by Λ.

We want the functionality of $\Lambda\Pi$ in the presence of unauthenticated networks to be as close as possible to the functionality of Π in the presence of authenticated networks. We consider two notions of 'closeness' in functionality. The first, somewhat weak, notion only requires that whenever the computation diverges from the way it would proceed in an authenticated network, at least *one* party will be aware of this fact, and will output some special `alert` signal to a higher layer or an operator. More precisely,

Definition 1 *An emulator Λ is t-aware if the following condition holds for any application protocol Π and any t-break-in-limited adversary $\mathcal{A}$. In any execution of $\Lambda\Pi$ in the presence of $\mathcal{A}$, whenever a message is not faithfully delivered, some party outputs* `alert` *in the same time-unit, except for negligible error probability.*[3]

Another "awareness property" may require that every "non-operational" party which is not broken-into will output an `alert` signal. We do not pursue this definition in this extended abstract. (Still the protocol we describe in the sequel has that property as well.)

The second notion of closeness to authenticated networks is much stronger. Here we require that, for any application protocol Π, running $\Lambda\Pi$ on an unauthenticated network will have *the same effect* as running Π on an authenticated network. That is, for any (limited) adversary $\mathcal{A}$ that operates against $\Lambda\Pi$ in the unauthenticated model, there should exist a (limited) adversary $\mathcal{B}$, operating against Π in the authenticated model, so that $\mathcal{B}$ uses comparable resources to $\mathcal{A}$, and has essentially the same effect on the computation as $\mathcal{A}$ does. (Essentially, $\mathcal{B}$ will 'simulate' the behavior or $\mathcal{A}$. This notion draws from definitions of secure multiparty computation [17, 1, 9].) We formalize this notion as follows.

[3] The error probability is determined by a security parameter. This parameter is implicit in our notations and description.

We first define the **adversary view** of an execution of some protocol to be all the data and communication seen by the adversary during this execution. Let $\text{VIEW}_{\Pi,\mathcal{A}}(\vec{x},\vec{r})$ denote the view of the adversary $\mathcal{A}$ when interacting with n parties running protocol Π on input $\vec{x} = x_1 \ldots x_n$ and random input $\vec{r} = r_1 \ldots r_n$ (x_i and r_i for party P_i). Also, let $\text{OUT}_{\Pi,\mathcal{A}}(\vec{x},\vec{r})_i$ denote the catenation of all outputs of party P_i while running protocol Π on input $\vec{x}$ and random input $\vec{r}$ in the presence of adversary $\mathcal{A}$, and let

$$\text{EXEC}_{\Pi,\mathcal{A}}(\vec{x},\vec{r}) = \text{VIEW}_{\Pi,\mathcal{A}}(\vec{x},\vec{r}), \text{OUT}_{\Pi,\mathcal{A}}(\vec{x},\vec{r})_1 \ldots \text{OUT}_{\Pi,\mathcal{A}}(\vec{x},\vec{r})_n$$

Finally, let $\text{EXEC}_{\Pi,\mathcal{A}}(\vec{x})$ denote the random variable describing $\text{EXEC}_{\Pi,\mathcal{A}}(\vec{x},\vec{r})$ where $\vec{r}$ is uniformly chosen.

Definition 2 *An emulator Λ is a (r,s)-*simulatable *if for any multiparty protocol Π, any $r' \leq r$, any $s' \leq s$, and any (r',s')-limited adversary $\mathcal{A}$ that controls an unauthenticated network, there exists a (r',s')-limited adversary $\mathcal{B}$, that controls an authenticated network, such that for all inputs $\vec{x}$ of the parties:*

$$\text{EXEC}_{\Pi,\mathcal{B}}(\vec{x}) \stackrel{s}{\approx} \text{EXEC}_{\Lambda\Pi,\mathcal{A}}(\vec{x}) \tag{1}$$

where $\stackrel{s}{\approx}$ stands for "negligible statistical distance", and the statistical distance is measured as a function of the security parameter.

Remarks:

1. In principle it suffices that the two sides of (1) be computationally indistinguishable. The only reason to require statistical proximity in (1) is that we can achieve this stronger security.

2. We define awareness with respect to break-in-limited adversaries, and define simulatability with respect to disconnection-limited adversaries. This choice was made simply because this is what our construction achieves. Conceivably awareness and simulatability can be also defined with respect to other types of adversaries.

3. Definition 2 requires that the authenticated adversary $\mathcal{B}$ does not break into or disconnect more parties than the unauthenticated adversary $\mathcal{A}$, even when $\mathcal{A}$ breaks-into or disconnects only a small number of parties. This provision 'counts out' trivial emulators that fail to transmit Π-messages on their own accord (such as the emulator that does not let some specific party to transmit any messages).

We can now state our main result:

Theorem 2 *There exists an authenticated network emulator that is both $(\lceil\frac{n}{2}\rceil - 1, \lceil\frac{n}{2}\rceil - 1)$-simulatable and $(\lceil\frac{n}{2}\rceil - 1)$-aware.*

3 Tools

In this section we present two tools used in our solution. We start by describing how we can obtain a "somewhat reliable" communication over an unauthenticated network. Next we describe our main tool: a *proactive, robust, distributed signature scheme.*

3.1 Communicating over faulty links

We describe a simple protocol that guarantees "somewhat reliable" communication over a network with faulty links. It is needed for the distributed signature scheme used in our construction. In our unauthenticated network we cannot guarantee that messages are received by all parties. We therefore only make sure that in order to prevent messages ¿from being delivered the adversary must tamper with many links. A party can participate in this protocol only if it has a certificate for its current verification key.

protocol Reliable-Send

1. *To send a message M, party P_i signs M using its current signature key, appends the current verification key along with the certificate, and sends it to all other parties.*

2. *Each party forwards the message to all intended recipients (we assume that M contains a list of the intended recipients).*

3. *Each party P_j checks all of the messages it got and accepts every message which is properly signed and certified, and is addressed to it.*

This simple protocol guarantees the following robustness property:

Observation 3 *If a party P has $\geq \lceil \frac{n}{2} \rceil$ faithful links to operational parties then*

- *Each message which is sent using* Reliable-Send *between two of these operational parties arrives at its destination.*
- *P receives any message which is sent to it by any of these parties.*

3.2 A distributed signature scheme

Loosely speaking, a **proactive distributed signature scheme** is a protocol by which a "global signing capability" is shared among a group of parties. That is, any sufficiently large subset of these parties are able to generate a signature on any message; still, as long as the adversary does not break into too many parties at the same time unit, she is unable to generate any new, valid message-signature pair. Signatures can be verified using a global, publicly available verification key.

A slightly more detailed description follows. A distributed signature scheme, resilient against some type of adversaries, consists of three parts: a key generation algorithm G, a multiparty signing protocol S, and a (personal) verification algorithm V. The key generation algorithm G generates a sequence $\vec{s} = s_1, \ldots s_n$ of shares of a signature key, together with a verification key v. Each party P_i out of n parties is given s_i (and v) at the initialization of the system. Given an input message m to be signed, the parties invoke protocol S to jointly generate a signature $S_{\vec{s}}(m)$, such that the verification algorithm with verification key v accepts the signature. That is, $V_v(m, S_{\vec{s}}(m)) = 1$. Furthermore, any adversary of the considered type should be unable to generate a new message m' and a signature x such that $V_v(m', x) = 1$. See [13] for a more complete definition.

In a **proactive** distributed signature protocol the parties engage, in each refreshment phase, in a protocol for refreshing their shares of the signature key. This is done in a way that prevents the adversary from generating fake signatures (and, in particular, from computing the signature key) as long as not too many parties are broken into at the same time unit. However, we stress that the global verification key must remain unchanged throughout. (This is the crux of our solution: since the verification key is unchanging, it can be kept in a ROM and can thus be protected against malicious modifications.)

Several proactive distributed signature schemes were proposed recently [19, 20, 13]. Also, in Section 5 we describe yet another such scheme, that is especially tailored for our needs. All these schemes are resilient to adversaries that break into at most $\lceil n/2 \rceil - 1$ parties at each time unit. (Parties broken into may diverge arbitrarily from their protocols.) However, all these schemes assume *authenticated networks,* whereas we have only an unauthenticated network. We deal with this problem as follows.

Dealing with unauthenticated links. We approximate authenticated communication by letting each message to be sent using the Reliable-send protocol. It follows from Fact 3 that, as long as any $(\lceil \frac{n}{2} \rceil - 1)$-operational party has a certificate, and at least $\lfloor \frac{n}{2} \rfloor + 1$ parties are $(\lceil \frac{n}{2} \rceil - 1)$-operational, any message sent between two $(\lceil \frac{n}{2} \rceil - 1)$-operational parties will be received unmodified. Consequently any protocol that assumes authenticated and reliable communication can be executed here.

An apparent problem arises here: The RELIABLE-SEND protocol uses certificates. However, as described in the Introduction, certificates are generated by the distributed signature protocol, which uses RELIABLE-SEND. We overcome this apparent 'circularity' in the next section.

We remark that the proactive distributed signature protocols described above use also private channels and broadcast channels. Such channels can be implemented on top of authenticated channels using standard encryption and agreement primitives.

4 An authenticated network emulator

Our emulator, denoted Λ_{cert}, consists of two parts: A refreshment protocol to be executed during each refreshment phase, and security transformations to be applied to the messages of the application protocol.

THE REFRESHMENT PROTOCOL. To bootstrap trust during refreshment phases, each party stores a verification key for some signature scheme in a Read-Only-Memory. This verification-key (which is the same for all parties) is installed in the parties at system set-up together with the emulator code itself, and remains fixed throughout the lifetime of the system.[4] This key should be thought of as a public key of a certification authority. In addition, at system set-up each party is given a share of the corresponding signature-key. This share *is stored in regular memory* and is changed at each refreshment phase, via the key refreshment protocols of the proactive distributed signature scheme. We refer to the fixed verification key and its corresponding signature key as the **global keys** of the system.

At the beginning of each time-unit, each party picks a new pair of signature and verification keys of some digital signature system. We refer to these keys as the **personal** keys of the party at that time-unit.[5]

Next, each party sends his new personal verification key to all other parties. The parties now invoke the distributed signature protocol to obtain a certificate for this key. The certificate is of the form "the personal verification key of P_i in time-unit ℓ is x". Note that, before the certificate is obtained, the party has no way to authenticate his messages. Thus he has no guarantee that the other parties indeed received the new personal verification key he just sent. However, once a certificate is obtained, the party can easily verify its validity using the global verification key in his ROM. If a valid certificate is not obtained then the party outputs `alert`. (In the analysis we show that, as long as the adversary is $(\lceil \frac{n}{2} \rceil - 1, \lceil \frac{n}{2} \rceil - 1)$-disconnection-limited, any $(\lceil \frac{n}{2} \rceil - 1)$-operational party always manages to obtain a valid certificate.)

Once valid certificates are obtained, the parties then execute the share refreshment protocol of the proactive distributed signature scheme, in order to refresh their shares of the global signature key.

AUTHENTICATING Π-MESSAGES. The obtained certificate is used to authenticate the application-protocol messages, as follows. Each message is signed using the current personal signature key; the current personal verification key is attached to the message, *along with the current certificate.* The receiving party will verify the certificate and then verify the signature using the attached verification key. If any verification fails then the receiver outputs `alert`. (Alternatively, each pair of communicating parties can use the certified keys to exchange symmetric session keys, say using [6], and use those to authenticate their communication.)

[4] Of course, this key can be replaced using manual intervention, which must be carried out in all parties during the same time-unit.

[5] Since selecting new signature and verification keys at each refreshment phase is required, it is preferable to use a signature scheme where choosing a new pair of keys does not require much work. In particular we suggest using an El-based scheme, where choosing new keys requires only one modular exponentiation.

Let Π be a protocol that assumes authenticated networks. We describe the combined protocol $\Lambda_{cert}P$. In each time-unit ℓ, each party P_i proceeds as follows.

Emulator $\mathbf{\Lambda_{cert}}$

(I) Refreshment-Phase.

1. *Choose a (private) signature key $sig_{i\ell}$, and a (public) verification key $ver_{i\ell}$, of an existentially unforgeable signature scheme.*
2. *Send the new verification key $ver_{i\ell}$ to all other parties. (Here we do not use* RELIABLE-SEND*, since parties may not have a certificate at this stage.)*
3. *Execute the distributed signature protocol (with threshold $t = \lceil n/2 \rceil - 1$) to obtain a certificate for every verification key $ver_{j\ell}$ which you got in Step 2. The communication during the distributed signature protocol is sent using* RELIABLE-SEND*. Then send the resulting certificates to all other parties. Do not participate in generating more than one certificate per party per time unit. (Only parties with certificates can send messages in this step. Still, even parties without certificates can receive the newly generated certificates.)*
4. *Execute the share-refreshment protocol (with threshold $t = \lceil n/2 \rceil - 1$) to refresh your share of the global signature-key. The communication during this protocol is done using* RELIABLE-SEND*. Parties can use either their new certificates or certificates from the previous time-unit to participate in this step.*
5. *If you failed to obtain a certificate for $ver_{i\ell}$ in Step 3, or failed to obtain a new share of the global signature key in Step 4, then output* `alert`*. Otherwise, the refreshment phase is successfully completed.*

(II) Security Transformations. *Execute the code of party P_i in Π with the following modifications:*

6. *Before sending each Π-message, sign it using the personal signature key $sig_{i\ell}$ and attache the personal verification key and the current certificate.*
7. *Upon receiving a message, verify that the messages is signed and certified properly. If the verification fails, ignore that message (and output* `alert`*).*

We analyze the construction in Appendix A.

5 An alternative distributed signature scheme

MOTIVATION. As described in Section 3, we can use several distributed signature scheme from the literature for the construction of Λ_{cert}. One drawback of these distributed signature protocols, however, is that they tend to have a high round and communication complexity. Moreover, since we use the RELIABLE-SEND protocol to implement each send instruction then the communication-complexity gets even worse. Below we describe a distributed signature protocol which may be more adequate for our needs.

The main advantage of our scheme is that almost all the work can be done off-line (i.e., independently of the message to be signed), where the on-line part (i.e., the part which depends on the message) is minimal, and in particular non-interactive. (Off-line . on-line work was first studied with respect to signature schemes in [11].) This property will be used in our scenario as follows. The off-line part of the signing process of a new personal verification key will be carried out *at the end of the previous refreshment phase.* Only the on-line part will be carried out at the beginning of a phase, when there may be no valid certificates.

In this scheme (like many others), each party's share of the signature key is used to generate a "signature part" for the given message. Then the parties collaborate to combined all these "signature parts" into one valid signature.

The main difference between this signature scheme and previous ones is the way we handle the signature-parts of faulty parties. Faulty parties may refuse to send their signature parts, or even send corrupted parts which may prevent us from combining the parts into a valid signature. In other schemes (e.g., [19, 20]) the correctness of each individual signature-part was verified before inclusion in the global computation. Instead, we make use of error-correcting codes to correct erroneous or missing signature parts. (This technique draws from [8].) That is, we construct the signature parts in a way that correct signature parts form a codeword. This way each party can use known error correction algorithms to correct the "bad signature-parts" (as long as there aren't too many of them), and then combine all signature parts into a valid signature. We stress that this a non-standard use of error-correcting codes, in that the 'word' to be corrected is a combination of messages sent from all parties, rather than a perturbed code-word sent by a single party.

Overview. The scheme is based on a known variant of El-Gamal signatures (which is also used in [19, 20]). As in any El-Gamal signature, the verification key is a triplet p, g, y, where p is a large prime, g is a generator in Z_p^* and y is another element in Z_p^*. The corresponding signature key is $x \in Z_p^*$ so that $g^x = y \pmod{p}$.

In this variant, the signature on a message m with respect to a public key p, g, y is a pair (r, s) so that $y^{H(m,r)} = g^s \cdot r^r \pmod{p}$, where H is an appropriate hash function. The signer who knows x (the discrete-log of y) can generate this signature by picking a random element $k \in Z_p^*$ and setting $r \leftarrow g^k \ (mod\ p)$ and $s \leftarrow x \cdot H(m,r) - k \cdot r \pmod{p-1}$.

We partition the signing protocol to an off-line and on-line stages. In the off-line stage the parties share both the secret x and the random number k among all parties (so each party P_i has shares x_i, k_i), and compute $r = g^k \pmod{p}$. In the on-line stage the message m is sent to all the parties, and each party computes its own signature-part s_i and sends (r, s_i) to the verifier. The verifier then applies an error-correction procedure to the s_i's to "weed-out" the bad parts and combines everything into one valid signature (r, s) with respect to p, g, y.

Sharing k and x. To share k, x the parties execute a protocol which is similar to the share-renewal protocol in [15]. This protocol uses Feldman's t-out-of-n Verifiable-Secret-Sharing procedure [12]. At the end of this protocol both the secret x and the random value k are shared among the parties in the following manner: There exists two t-degree polynomials $X(\cdot)$ and $K(\cdot)$ so that

- $x = X(0) \pmod{p-1}$ and $k = K(0) \pmod{p-1}$; and
- Each party P_i has shares $x_i = X(i) \pmod{p-1}$ and $k_i = K(i) \pmod{p-1}$.

An artifact of using Feldman's procedure is that at the end of the protocol each party also knows the value $r = g^k \pmod{p}$.

Computing signature parts. Given a message m, each party P_i (who knows k_i, x_i and r) sets $s_i \leftarrow x_i \cdot H(m,r) - k_i \cdot r \pmod{p-1}$. The signature-part of party P_i is (r, s_i).

Notice that since m, r are the same for all parties (and hence, so is the value $H(m,r)$), then there exists a t-degree polynomial $S(\cdot) = H(m,r)X(\cdot) - rK(\cdot)$ so that $s_i = S(i) \pmod{p-1}$ for all i. Thus the vector of s_i's form a codeword over a Generalized-Reed-Solomon code.

Combining the signature parts. Upon receiving all the signature-parts, the verifier first finds the majority value for r. Then it takes the vector of s_i's (some of which may be corrupted), and applies to it the Berlekamp-Welch decoding procedure [4]. This procedure is guaranteed to compute the coefficients of the polynomial $S(\cdot)$ as long as no more than $(n-t)/2$ of the s_i's are corrupted. After finding $S(\cdot)$, the verifier computes $s \leftarrow S(0) \pmod{p-1}$. Notice that (r, s) is a valid signature on m with respect to p, g, y, since

$$s = S(0) = H(m,r)X(0) - rK(0) = x \cdot H(m,r) - k \cdot r \pmod{p-1}$$

and therefore $y^{H(m,r)} = g^{x \cdot H(m,r)} = g^{s+k \cdot r} = g^s r^r \pmod{p}$.

Remark. One drawback of this scheme is that the Berlekamp-Welch decoder can correct at most $(n-t)/2$ corrupted parts (where t is the degree of the polynomials in the Generalized-Reed-Solomon code). Since the degree of the polynomial in our construction is $\lceil \frac{n}{2} \rceil$ then our scheme will be at most $(\lceil \frac{n}{2} \rceil - 1, \lceil \frac{n}{4} \rceil - 1$-simulatable. (The awareness remains the same.)

References

[1] D. Beaver. Foundation of secure interactive computing. *Proceedings of CRYPTO'91.*

[2] Mihir Bellare and Phil Rogaway. Entity authentication and key distribution. *Proceedings of CRYPTO'93,* pages 232–249, August 1993.

[3] Mihir Bellare and Phil Rogaway. Provably secure session key distribution — the three party case. in *proceedings of STOC'95*, pages 57–66.

[4] E. Berlekamp and L. Welch Error Correction of Algebraic Block Codes. US Patent Number 4,633,470.

[5] Ray Bird, Inder Gopal, Amir Herzberg, Phil Janson, Shay Kutten, Refik Molva, and Moti Yung. Systematic design of two-party authentication protocols. In *proceedings of CRYPTO 91*, August 1991.

[6] W. Diffie, P. C. Van Oorschot and M. J. Weiner. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography 2*, pages 107-125, 1992.

[7] Ray Bird, Inder Gopal, Amir Herzberg, Phil Janson, Shay Kutten, Refik Molva, and Moti Yung. Systematic design of a family of attack-resistant authentication protocols. *IEEE Journal on Selected Areas in Communications*, 11(5):679–693, June 1993. Special issue on Secure Communications.

[8] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. *proceedings of 20th STOC,* pp. 1-10, 1988.

[9] R. Canetti. Studies in secure multiparty computation and applications. Ph.D. thesis, The Wiezmann Inst. of Science, 1995.

[10] R. Canetti and A. Herzberg. Maintaining security in the presence of transient faults. In *Proceedings of CRYPTO'94.*

[11] S. Even, O. Goldreich and S. Micali. On-line/off-line digital schemes. In proceedings of *Advances in Cryptology CRYPTO'89*, G. Brassard, editor. Lecture Notes in Computer Science, vol 435, Springer-Verlag, New York, 1990. Pages 263–277.

[12] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In proceedings of *28th FOCS*, IEEE, 1987.

[13] R. Gennaro, S. Jarecky, H. Krawczyk and T. Rabin. Robust Threshold DSS Signatures. *Proceedings of EUROCRYPT 96,* to appear.

[14] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the* 19^{th} *STOC, New York City*, pages 218–229, 1987.

[15] A. Herzberg, S. Jarecki, H. Krawczyk and M. Yung. Proactive Secret Sharing or: How to Cope with Perpetual Leakage. In *Proceedings of CRYPTO'95.*

[16] P. Karn and W. A. Simpson, "The Photuris Session Key Management Protocol", Internet Draft draft-ietf-ipsec-photuris-09.txt, January 1996.

[17] S. Micali and P. Rogaway. Secure Computation. In *Proceedings of CRYPTO'91.*

[18] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the* 10^{th} *Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada*, pages 51–59, 1991.

[19] S. Jarecki. Proactive Secret Sharing and Public Key Cryptosystems. M.Eng. thesis, MIT, 1995

[20] Proactive public key and signature systems. *submitted to this conference*

A Security of Λ_{cert}

We first prove a simple lemma:

Lemma 4 *At each refreshment round ℓ, and for each party P_i, at most one certificate of the form* "the personal verification key of party P_i ar time unit ℓ is ..." *is generated.*

Proof: At least $\lfloor\frac{n}{2}\rfloor + 1$ parties are required to generate a certificate. Thus, if two certificates of the above form are generated then there exists a party that perticipated in generating both certificates. However, the code of Λ_{cert} does not allow this option. □

Theorem 5 *Λ_{cert} is a $(\lceil\frac{n}{2}\rceil - 1)$-aware, authenticated network emulator.*

Proof (sketch): Assume a message of protocol Π, sent from P_i to P_j, is not faithfully transmitted, and no party outputs `alert` at this time unit. Since P_j does not output `alert` at Step 7 of Λ_{cert}, then the adversary managed to produce a fake verification key for P_i for this time unit, together with valid certificate. From the security of the proactive distributed signature scheme it follows that this fake pair must have been generated via an application of the signature protocol. However, Lemma 4 states that at most one such cetificate is generated for P_i for this time unit. Therefore it must be that party P_i did not get a certificate for his real verification key at this time unit, and has output `alert` at Step 5. □

Theorem 6 *Λ_{cert} is a $(\lceil\frac{n}{2}\rceil - 1, \lceil\frac{n}{2}\rceil - 1)$-simulatable authenticated network emulator.*

Proof (sketch): We first prove the following lemma.

Lemma 7 *For any application protocol Π, and any adversary $\mathcal{A}$, it holds that in the combined protocol $\Lambda\Pi$, If P_i is $(\lceil\frac{n}{2}\rceil - 1)$-operational in time-unit ℓ then it will obtain a certificate and a refreshed share of the global signature key for this time-unit.*

Proof (sketch): We use the induction hypothesis and Fact 3 to argue that if P_i is $(\lceil\frac{n}{2}\rceil - 1)$-operational in time-unit ℓ then there are at least $\lceil n/2\rceil$ parties with certificates and shares from time-unit $\ell - 1$ such that

(a) they all receive $ver_{i\ell}$ and try to sign it at the beginning of time-unit ℓ,
(b) P_i receives every message which is sent to it from any of these parties.
The validity of the signature and the share-refreshment protocols implies that both these protocols succeed from P_i's point of view. Hence P_i obtains both a certificate for its new personal verification key and a new share of the global signature key. □

Corollary 8 *Let* Π *be any protocol,* $\mathcal{A}$ *be any* $(\lceil\frac{n}{2}\rceil-1,\ \lceil\frac{n}{2}\rceil-1)$*-limited adversary that operates against* $\Lambda\Pi$ *in the unauthenticated model. Then the probability of the event in which at some time-unit in the execution of* $\Lambda\Pi$ *there exist two parties* P_i, P_j *so that*

- *Both* P_i *and* P_j *are* $(\lceil\frac{n}{2}\rceil-1)$*-operational,*
- P_i *accepts a* Π*-message as coming from* P_j*, but*
- *This message was not faithfully delivered*

is negligible (the probability is taken over the coin tosses of all parties and $\mathcal{A}$ *and measured against the security parameter of the system).*

Proof (sketch): Recall that $\Lambda\Pi$ verifies the signature and certificate on every Π-message before it accepts it. It follows from Lemma 4 that if P_j is $(\lceil\frac{n}{2}\rceil-1)$-operational then the only certified verification key for P_j in this time-unit is the one chosen by P_j itself. Thus the adversary does not know this key and cannot forge messages with respect to it (except with negligible probability). □

Once we have Corollary 8, proving the rest is not hard. We need to show that there is an adversary $\mathcal{B}$ in the authenticated model that emulates $\mathcal{A}$ in the unauthenticated model. The adversary $\mathcal{B}$ has access to Π in the authenticated model and it tries to simulate an execution of $\mathcal{A}$ with $\Lambda\Pi$.

To do that, $\mathcal{B}$ uses its access to Π and implement the emulation part Λ by itself. As long as the event in Corollary 8 does not happen, $\mathcal{B}$ can continue the simulation, playing the roles of the operational parties in the refreshment phases, faithfully delivering messages which $\mathcal{A}$ faithfully delivers and omitting all other messages. The view of $\mathcal{A}$ in the simulated execution is identical to its view in the real execution.
□