The Proactive Security Toolkit and Applications

Boaz Barak, Amir Herzberg, Dalit Naor and Eldad Shai

IBM Haifa Research Lab, Tel-Aviv Site
E-Commerce and Technologies Group
{barak,amir,dalit,eldad@haifa.vnet.ibm.com}

`You can't cheat all people at the same time.` -- (anonymous)

Abstract

Existing security mechanisms focus on prevention of penetrations, detection of a penetration and (manual) recovery tools. Indeed attackers focus their penetration efforts on breaking into critical modules, and on avoiding detection of the attack. As current security mechanisms are imperfect and existing systems often implement them poorly, it is well recognized that attacker are in fact able to penetrate. Security tools and procedures may cause the attackers to lose control over a specific module (computer, account), since the attacker would rather lose control than risk detection of the attack. However, while controlling the module, attacker may learn critical secret information or modify the module (plant trapdoors) that make it much easier for the attacker to regain control over that module later, or to defeat the overall security goal (even without regaining control of that module).

Recent results in cryptography give some hope of improving this situation. These results show that many fundamental security tasks can be achieved with *proactive security*. Proactive security does not assume that there is any module completely secure against penetration. Instead, we assume that at any given time period (day, week,...), a sufficient number of the modules in the system are secure (not penetrated). The results obtained so far include some of the most important cryptographic primitives such as signatures, secret sharing, and secure communication. However, there was no usable implementation, and several critical issues (for actual use) were not addressed.

In this work we report on a practical toolkit implementing the key proactive security mechanisms. The toolkit provides secure interfaces to make it easy for applications to recover from penetrations. The toolkit also addresses other critical implementation issues, such as the initialization of the proactive secure system. We describe the toolkit and discuss some of the potential applications. Some applications require minimal enhancements to the existing implementations - e.g. for secure logging (especially for intrusion detection), secure end-to-end communication and timestamping. Other applications require more significant enhancements, mainly distribution over multiple servers; examples are certification authority, key recovery, and secure file system or archive.

1. Introduction

Traditional security systems assume that one or more systems are always secure, i.e. are never controlled by the attackers. The model of Proactive Security does not make this assumption. Instead, it considers cases where all components of the system may be broken-into and controlled by an attacker, with restrictions on the number of components broken-into during the same time period (day, week,...).

Proactive security shows how to maintain the overall security of a system even under such conditions. In particular it provides automated recovery of the security of individual components, avoiding the use of expensive and inconvenient manual processes (except for some `aggressive` attacks, which cannot be prevented - but are definitely and clearly detected). The technique combines two well-known approaches to enhance the security of the system: *distributed (or threshold) cryptograph*, which ensures security as long as a threshold (say half) of the servers are not corrupted (see [12]); and *periodic refresh (or update)* of the sensitive data (e.g. keys) held by the servers. In short,

proactive = distributed + refresh

This way, the proactive approach guarantees uninterrupted security as long as not too many servers are broken into at the same time. Furthermore, it does not require identification when a system is broken into, or after the attacker loses control; instead, the system proactively invokes recovery procedures every so often, hoping to restore security to components over which the attacker lost control.

Proactive security is highly desirable in many realistic settings, in particular:

- When a high level of security is required, together with fault tolerance (as redundancy improves fault tolerance but opens more points for attack)
- To ensure acceptable level of system security using weakly secure components such as most commercially available operating systems.

(Examples of specific applications are given below.)

Recent results show that many fundamental cryptographic functionalities may be achieved even under the proactive security model - as long as most components are secure most of the time. In particular, proactively secure protocols have been devised for the following problems:

- Secret sharing [21,16]
- Discrete-log-based digital signatures [15], and in particular DSA [13]
- Secure end-to-end communication [5]
- RSA [10,11,24], and in particular generation of the RSA shared key [3]
- Pseudo-random generation [6,8]
- Key distribution center [20]

This substantial set of known results in proactive security did not yet produce any practical security product or solution. (In fact, there are only a few deployments of distributed security - the most well known may be the SET credit card standard's certificate authority [7]; see also `related works` below.) The creation of such a proactive solution is non-trivial, as the protocols are often quite complex and nontrivial to implement. Furthermore, the protocols are specified under some simplifying assumptions and do not address some needed elements, such as interfacing between the proactive service and the applications using it. This paper reports on a toolkit, to be soon placed for public experimentation, to allow practical deployment of proactive security. The main new contributions are:

- *A secure initialization mechanism*, with reasonable, practical requirements from the computer and operating system. Specifically, all we require is a secure boot process (which is a good idea anyway, against viruses - and easily done with signed code); and a per-machine secret-private key pair, with the public key protected from modification (e.g. in ROM or write-once EEROM), and the secret key in erasable memory (e.g. disk). Previous results required storage of parameters specific to the particular application (such as the group's public key) in secure storage, which is not practical.
- *A set of application program interfaces (APIs)* that allow the use of the toolkit to improve security, specifically provide security in spite of break-ins into computers, of existing applications, as well as the development of new applications which are proactive secure.

The security of any proactive solution relies heavily upon its correct architecture and integration with existing, non-proactive, operating system. The design of our toolkit, which does not view the proactive model as series of protocols but, rather, as a security enhancement of the operating system which transforms it into a proactively secured system via the appropriate use of proactive protocols, has not been defined nor implemented in the past. We show that it is possible to transform general applications which are required to remain secure for long periods of time to operate in a proactive environment, namely *proactivizing* applications. Specifically, we show how to appropriately use the proactive cryptographic functions as key primitives in the proactivization process. To this end, we define an architecture for a **proactive operating environment** which serves as a platform on which standard applications can be proactivized. This operating environment consists of a network of servers which is set up once, which we call the *proactive network*. Each server is instantiated at boot time by the operating system and is checked periodically, also by the operating system. Servers can recover data (both public and private data) from other servers in the proactive network, if such data is corrupted or lost. Once the proactive network is set up, any application can run on the top of the network and request proactive services by the means of API. The feasibility of the proactive model and of the architecture presented hereby has been demonstrated by the *Proactive Security Toolkit*, which is a Java implementation described in this manuscript.

1.1 Applications of the Proactive Security Toolkit

There are three kinds of applications that may take advantage of the proactive security toolkit to recover from penetrations:

- **Centralized applications** - a `traditional` application running on one server only. The application uses a proactively secure service provided by the toolkit. For some applications and services, this could provide significant advantage - at minimal change to existing applications. Some typical applications are:
  - **Secure logging:** each client application may add entries (events) to the log; however none of them can modify or erase the log. This could be of great value in improving intrusion detection tools, as intruders often try to erase traces in log files.
  - **Secure end-to-end communication:** the proactive toolkit can provide the applications with freshly generated and certified public keys periodically. This could be integrated with tunneling mechanisms such as secure IP or SSL.
  - **Timestamping:** the toolkit could be used to sign a document (or its hash) and current time, to prove that the document existed at this time.
- **Distributed applications** (Figure 1) - the application runs simultaneously on all machines (App_1, ..., App_n) and requests services through all machines. Each App_i interacts directly with its own proactive server (PS_i). A typical application is a **certificate authority**, or in general any workflow application requiring secure (multi-person) digital signatures. Another application is **key recovery** (escrow agents).

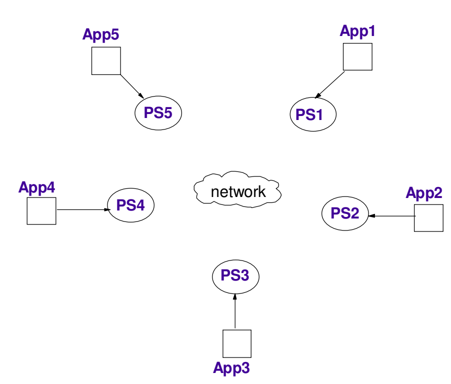

Figure 1: Distributed or Proactive Application of the Toolkit

- **Proactive applications** - the application runs in a distributed configuration but, in addition, goes through periodical refreshes by utilizing the proactive toolkit services. This is required when the application security or efficiency requirements cannot be met by the services of the toolkit. Examples include multiparty protocols such as voting and trading, database, operating system, and access control mechanisms. An especially interesting application is a **Secure Commerce Server** - such server can not lie within the firewall although it handles confidential data and matters (such as access control, certificates etc.). It is therefore natural to proactively distribute the server among a number of (independent, and possibly not even mutually trusted) hosts and locations, thus achieving increased trust in the server.

1.2 Related Systems

A number of distributed systems related to the proactive model have been designed and implemented. The Intrusion Tolerance via Threshold Cryptography (ITTC) project [17,19,29] designs and builds tools and an infrastructure that are based on Threshold Cryptography, and use an intrusion tolerant Certification Authority and an intrusion tolerant web server to demonstrate these tools. The technical approach is based upon a distributed RSA key, such that *k* “share servers” generate a private RSA key that is shared among them from the moment of creation, and any *t* of the share serves can be used to apply the key. Another related system is the Omega Key Management Service system [27], designed and developed at Bell-Labs. The Ω system is a distributed public key management system. It employs threshold techniques which can tolerate a number of server failures (via the implementation of the Rampart toolkit [25]), but not recovering (proactive) techniques. The *e-Vault* (electronic-Vault) project [18] of IBM is an implementation of a distributed data repository. It employs RSA-based shared signatures as the basic cryptographic function.

There are a few implementation efforts of proactive algorithms. Specifically, the Network Randomization Protocol (NRP) of [8], which provides a proactive pseudo-random generator, has been implemented at IBM. It also provides a simple API for client applications to get pseudo-random values from the servers. Another effort, the implementation of proactive threshold key protocols, has been reported in [14].

1.3 Organization of the paper

This paper is organized as follows. In Section 2 we describe the model as well as an overview of related work. Section 3 discusses the basic architecture of the proactive server. Sections 4 presents the protocol that initializes the network of servers and handles refresh/recovery of a server. The API module, which provides mechanisms for proactivization of applications, is fully described in Section 5. Java-related and other implementation issues, user-interface are the focus of Section 6.

2. Overview of the Proactive Model and Algorithms

2.1 Model

The proactive model assumes a set of *n* servers, *{P1, P2, ... Pn},* that are interconnected by a complete point-to-point communication channels. **Time** is divided into periods (like days, weeks, ...) which are determined by some global clock. An **adversary** may (temporarily) attack up to *t* of the *n* servers at any given time period - but at different time periods, different sets of *t* servers can be attacked. As a result, all servers engage in a refreshment stage at the beginning of each time period, so that any servers which has been attacked during past periods may automatically recover from possible undetected break-ins. Corruption is assumed to be either static (for example, disconnect a server from the rest of the network, eavesdrop, read secret data) or active/malicious (for example, deviate from the protocol, corrupt local data etc.) Therefore, after the attacker loses control over a server, the attacker may still know secret information of that server (e.g. passwords or secret keys). Furthermore, before losing control, the attacker may have corrupted (modified) some of the server's data (e.g. public keys of certificate authorities). The refreshment stage deals with both aspects, i.e. recovers any corrupted data and invalidates any old secret data (by choosing new secrets or splitting global secrets into a new set of shares). This brings the server back to a running stage, and guarantees that any information that was gathered by the adversary becomes worthless after recovery.

The fact that we limit the attacker to t corruptions, out of n servers, is similar to the distributed (or threshold) security model used in many works in distributed computing and cryptography. However, in the proactive security model we allow the attacker to corrupt *every* server - as long as it does not corrupt more than t servers *at the same period.* We say that adversary in the proactive model is mobile, namely attacked components may be released at some point (due to some security measure or other change in the system or the adversary causing loss of control, often as a result of an attempt by the adversary to avoid detection of the attack). Furthermore, in contrast to other approaches, proactively secure systems do not wait until a break-in is detected. Instead, a proactively secure system invokes the refreshment protocol periodically (and proactively) in order to maintain uninterrupted security, or force detection. For more discussion on the motivation behind this model, see [4,5,6,16].

Some attacks on the system cannot be prevented. The `classical` example is if the attacker is breaking into a server, thereby finding all its secret keys; it then pretends to be that server while keeping this server disconnected from the other servers (when the attacker lost control over that server). However, in such cases we will be able to *detect* the attack, and raise an *alert* - inform the operator about the attack. Operators will normally respond to such an alert by invoking special emergency security resources and procedures, which are very likely to remove the attacker - and possibly catch her as well. Therefore, it is highly unlikely that (smart) attackers will use such `visible` attacks.

The proactive security model assumes that even during attack, some specific data cannot be corrupted. The obvious example for data that we must assume cannot be corrupted is the program itself; if it could be changed, recovery is clearly impossible. Clearly, the program is not any different than any constant value used by the program; we will therefore assume that each computer comes with a read only memory which we can specify its contents. Specifically we assume that each computer comes with such a read only memory containing a fixed public key, and the corresponding secret key is known only at initialization - for a more detailed discussion, see Section 3.2. This assumption is not too difficult to implement in practice.

2.2 Toolkit's Functionalities and Algorithms

Our toolkit maintains two basic proactive functionalities for the entire lifetime of the system, as long as there are 'enough' working components in the system:

1. Proactively secure end-to-end communication (authenticated and encrypted) among all the nodes of the proactive networks, that is, new communication keys are agreed upon at the beginning of each period. This functionality is achieved using the protocols of [5].
2. A distributed signature key that is generated at initiation of the proactive environment, shared among all servers of the network and proactively maintained so that private shares are refreshed periodically without changing the signature public key. This internal signature key is often used for 'group certification' purposes and, for example, is mandatory for the implementation of the proactive end-to-end communication. Our toolkit implements a DSS distributed key using the algorithms of [15,13], but it is also possible to use a distributed RSA key, based on the signature algorithm of [24,11] with the key generation algorithm of [3].

The implementation of these functionalities are based on a number of algorithmic which, for completeness, are briefly outlined in the Appendix.

3. The Proactive Toolkit Architecture

Recall that the **proactive operating environment** serves as a platform on which standard applications can be proactivized. In this section we define the basic architecture and functional components for such environment.

The proactive operating environment consists of a network of servers which is set up once - this network is referred to as the Proactive Network [Figure 1]. Each node in the network runs a proactive server (**PServer**), whose basic architecture is depicted in Figure 2: A Pserver communicates with other Pservers via the proactive network, and provides proactive services to applications by the means of API. A server is instantiated at boot time and checked periodically by the operating system.

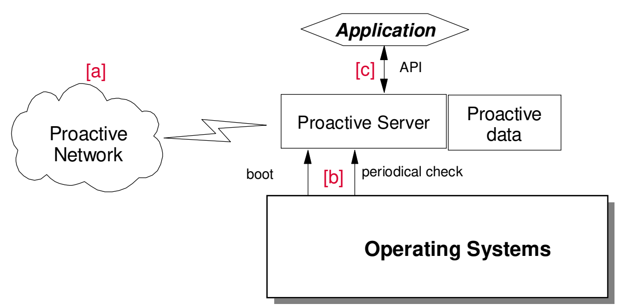

Figure 2: Architecture

3.1 Modules of the PServer

The internal design of a Pserver is composed of the following modules:

- **The Controller** This is the main "engine" of the program. It manages the Pserver data, dispatches incoming messages to appropriate protocols in the protocols tree (to be discussed below), keeps the status of the proactive network (namely, which nodes are active and running).
- **Communication Module** This module is responsible for the secure transmission and receipt of messages across the proactive network and with the various applications that use the Pserver services.
- **Library of Proactive Utilities** This library is built upon a standard cryptographic library and is a collection of utilities (objects and methods) that are needed for implementation of proactive algorithms and protocols. It includes Shamir's secret sharing (SS), Feldman's Verifiable Secret Sharing, Joint Secret Sharing and error-correcting polynomial interpolation. Note that this library consists of the functionalities that are needed *locally* in order to perform the protocols.
- **Library of Proactive protocols** A proactive protocol (in contrast to a proactive utility) is a thread of code that is executed at one server and performs the logical flow of communication steps required by a certain protocol, using the proactive utilities library for its local computations. The proactive toolkit is essentially a collection of protocols that are executed by the Pservers (either for its proper operation or as a proactive service to other applications). This module, probably the most important module in our toolkit, is a library of a number of protocols. Examples include: all

variants of Secret Sharing protocols, the Init, Recover and Refresh protocols of the server (Section 4), DSS_KeyGeneration, distributed_DSS protocol and DSS_KeyRefresh, Refresh_KeyTable protocol and a broadcast protocol.

- **The API Module** A module that provides interface to applications that run over the proactive operating environment. Section 5 describes this module in detail.

3.2 The Pserver Data

The Pserver, as any other program, maintains some key internal data. However, the maintenance of this data raises few algorithmic problems, as the server must be able to refresh and recover itself periodically, and this includes recovering its data or at least verifying that it has not been corrupted. The server's data is one of three types:

1. ROM data - this "write once" data is assumed to be immutable so that any attack on the system can not tamper with it, however an adversary may learn it. It is used for bootstrapping purposes as otherwise a recovering server could not bring itself to a secured state. Our design, as detailed in Section 4, attempts to minimize the amount of data that must be stored in the ROM in order to safely boot the server; in particular it shows that it suffices to store one public key (in our specific implementation, the server's port number as well) in the ROM for the Pserver to be completely recoverable.
2. Public data. Parts of this data are common to all servers, but other parts are specific to the particular Pserver, yet its exposure to the entire proactive network does not interfere with the security of a PServer. Since this data is necessary for the proper operation of any server and thus must be recoverable, it is duplicated the data among all servers so that during recovery it can be reconstructed if needed with the assistance of the proactive network. The details of this process are described in Section xx. The public data may be extended during the lifetime of the system, for example by generating new long-lived secrets (the common fields of these long-lived secrets is added to the public information).
3. Private data, specific to a particular server. One such example is the server's share of a distributed key. This data is typically not recovered, but instead is refreshed. It also requires the ability to be completely erased from the system without leaving any traces, which is a property that needs to be supported by the operating system.

4. The Proactive Toolkit Protocols

Our suggested design for a proactive operating environment must maintain proactively secure communication among the servers as well as a proactive internal signature key for the entire lifetime of the system. For that, these two protocols must be initiated and undergo refresh at every period, where a refresh may actually involve recovery at some server if it had detected that some of its data was corrupted or lost. We follow [5] for the design of the integrated proactive protocols of signatures and secure communication, and [13,15] for the specific proactive signature mechanism (Appendix A.3). However, [5] requires every server to keep in read-only memory (ROM) a copy of the verification key *Vcert* of proactive system (whose corresponding secret key *Scert* is shared between all the proactive servers, and these shares $S^i_{cert}(t)$ are refreshed at every period *t*). This assumption is not very practical, as the proactive system's key *Vcert* is not available when the computer is manufactured and sold, but only much later - when it is integrated into a specific proactive environment. We show how to provide the requirements of [5] while requiring only that each computer comes with pre-installed, machine-unique pair of secret key S^i_{start} (on erasable disk) and public key V^i_{start} (on ROM).

Another practical aspect which we had to deal is that the proactive server needs some constants configuration information such as IP addresses of other servers, cryptographic parameters used in the cryptographic algorithms, and so on. We denote this set of these (public) constants by *C*. Our protocols include mechanisms to recover C periodically (if the adversary corrupted C when breaking into the server at the previous period). Let $Mi=[S^i_{start}(V_{cert},C)]$ be the signature of server *i* on (V_{cert},C) using its initial key S^i_{start}. We denote by *M* the concatenation of all *Mi*'s, that is $M = (M_1,M_2, ...M_n)$. Hence, *M* is the *Invariant Information* of the system.

We begin by briefly reviewing the periodical refresh protocol of [5], which assumes the availability of an unmodified *Vcert* at every proactive server. We then describe the *Vcert-recover* protocol, with a *periodical-recover module* which recovers *Vcert* at the beginning of every refresh period (before using [5]), and an *initialization module* that uses $(S^i_{start},V^i_{start})$.

4.1 The Refresh Protocol of [5] for Period t

The goal of this sub-protocol is to refresh the communication/authentication keys as well as the shares of all long-lived keys, including *Scert*. A server which participates in this protocol may be "operational", in which case it has a valid pair of keys *(Si(t-1),Vi(t-1)), (Ei(t-1),Di(t-1))* and valid shares of the long-lived secrets (including $S^i_{cert}(t-1)$) from period *t-1*. Alternatively, a server can be "recovering" so that all of the above information is missing (or corrupted); even in this case we can assume that server posses *Vcert* and the constants *C* (to be ensured by the *Vcert-recover* protocol).

Perform a standard key refresh for period *t*:

1. Generate *(Si(t),Vi(t))*
2. Generate *(Ei(t), Di(t)),* Sign with *Si(t)*
3. Broadcast *Ki = (Vi(t), Si(t)((Ei(t)))* (using *Si(t-1)* when available) [1]
4. Generate a distributed signature *Scert(KeyTable)* where *KeyTable = [K1, ... , Kn].* [2]
5. Verify the signature *Scert(KeyTable).* [3]

(All channels are now authenticated/encrypted with refreshed keys of period *t*.)

Refresh long-lived data (S^i_{cert}):

1. Perform an agreement protocol on the data that needs to be refreshed/recovered
2. For any long lived secret *S* [4]
   (i) Reconstruct the missing shares of the secret *S* for the recovering servers
   (ii) Engage in a standard *Refresh (S)* protocol

Refresh Protocol for period t

Remarks:

1. Step 3 of the standard key refresh allows the option of sending *Ki* in the clear, since a recovering server has no valid *Si(t-1)* at this point. Hence, the following judgment should be made:
   i. if more than one (but different) authenticated messages arrive from server *i*, discard all of them.
   ii. if one authenticated message arrives, but few others in the clear, accept authenticated.
   iii. if more than one (but different) messages arrive in the clear, discard all of them
2. Recovering servers do not take part in the Joint-signature generation.
3. When verifying the signature *Scert(KeyTable)* server *i* must verify that *KeyTable* contains *Ki* (this serves as a "random challenge" to avoid replay of signature). If server *i* discovers that *Ki* is not part of the signed *KeyTable,* then raise an alert for a detected attack.
4. Use the secret recovery and refresh algorithms for proactive secret sharing, as described in [16] (see also Appendix A.1).

4.2 The *Vcert-recover* Protocol - *periodical-recover module*

The *periodical-recover module* is invoked at the very beginning of very refresh period, and re-generates *Vcert* and the constants *C* for any server which lost this data. As a result, it brings a recovering server to a state from which it can participate in the Refresh protocol described above. We assume that any operational server has a valid copy of a signature on *M*, the Invariant Information of the system, signed by the distributed signature key *Scert* - an assumption that is justifies by the *initialization module* described next.

Essentially, this protocol allows any recovering server to gather *M*, the Invariant Information of the system, from other operational servers as long as there are enough of them. Note that *M* needs to be "pushed" around the system since a recovering server may not know who its partners are (recall that *C*, the program constants, contains information such as IP addresses). The protocol is executed by all servers, and by the end of it a server detects whether it is "operational" or "recovering".

PushM algorithm of server *i*

Read V^i_{start} from ROM and check validity of the signature *[Scert(M),M]* stored in a file:

- **(i)** Extract *Mi* from *M* and check validity of signature of *Mi* with V^i_{start}.
- **(ii)** If signature on *Mi* is valid, obtain *Vcert* from *Mi* and validate signature *Scert(M).*

If signature on *M* is valid - **server *i* is operational**

- Send *M* to all other servers

Otherwise (*M* is not valid or file does not exist) - **server *i* is recovering**

- Wait until a verifiable copy of *[Scert(M),M]* arrives from other servers (otherwise raise ALERT - recovery has failed).

The *Vcert-recover* Protocol - *periodical-recover module*

4.3 The *Vcert-recover* Protocol - *initialization module*

This protocol is executed at the setup of the system. Its goal is to bring the servers to a state from which they can safely perform the *periodical-recover module* at every Refresh stage and achieve proper operation of the system. The protocol does the following: it first generates the initial set of authentication/encryption keys of the system, it generates the distributed signature key *(Vcert, Scert)* and finally produces a joint signature *[Scert(M),M]* on the Invariant information to help recovering servers bootstrap their data in the future. The input to this protocol is *C*, the program's constants, and $(S^i_{start}, V^i_{start})$ where V^i_{start}, the public part of this key, is also written in the ROM.

Input:$(S^i_{start}, V^i_{start})$, C

(Communication is secure - inactive adversary). Broadcast keys:

- Generate $(Si(0),Vi(0))$
- Generate $(Ei(0), Di(0))$, Sign $[Si(0)(Ei(0))]$
- broadcast (in clear) $(Vi(0), Si(0)(Ei(0)))$ to all servers

(Channels are now authenticated and encrypted. All servers must be cooperative, all communication channels must be operational!)

✓ Generate *(Scert,Vcert):*

- Engage in the generation of a distributed signature key *(Scert,Vcert)*. Server *i* gets $[Vcert, S^i_{cert}]$.

✓ Generate *(Scert(M),M):*

- Sign $Mi=[S^i_{start}(Vcert, C)]$
- Erase .S^i_{start} (VERY IMPORTANT)
- Broadcast *Mi* to all and receive *Mj* from all *j*. Construct the *InvariantInfo* $M=(M1,...,Mn)$.
- Engage in a generation of a distributed signature to generate *[Scert(M),M]*

(Channels authenticated/encrypted. No further assumptions on adversary behavior)

Initialization Protocol of server *i*

5. The Application Program Interface (API) Module

This section describes the interface between the proactive toolkit and the applications using it. A centralized application runs on the same computer running the proactive server; distributed or proactive applications (Figure 1) run one instance of the application on each of the proactive servers. The goals of the API are to provide secure communication between the application and the server. We will assume that the operating system is providing basic security services which allow server and client to restrict communication to the same computer, and to separate between two applications. There is one element of security that we must add in the API, which is, how to identify multiple instances of the same application running on the different servers - this will be done with a secure registration mechanism. The API's categories are:

1. Registration API's (must be used first, to get a handle to be used for other API calls)
2. Data Storage API's
3. Communication API's
4. Service API's

5.1 Registration API's

Since a Pserver can possibly service many clients, it is necessary to provide an authentication mechanism for requests, by which a request is uniquely associated with the client application it originated from (the application's "name"). For example, if an application by the name of "VeriSign_CertificateAuthority" is registered at the proactive server, asks for a generation of a proactive signature key and then repeatedly asks to proactively sign messages by this key, then the server needs to authenticate these requests. The registration mechanism is designed to address exactly this problem. Registration will provide the application with a *handle*, which it will append to every subsequent request. Different registration mechanisms are needed for different client configurations (centralized vs. distributed/proactive).

Centralized Application - this is the straightforward registration, designed to support requests that are initiated by a single client. The client sends *Registration_Req(Name, UniqueID)* where *UniqueID* is some random identifier of the application; the server responds with a handle.

The server needs to assign a quota on the number of services a specific client can request.
Distributed Application - The group of Pservers need to identify clients with the same name running on different machines. In particular, a request will be serviced by the proactive servers only if a majority of authenticated clients have initiated (or approved) this request. We suggest two options:

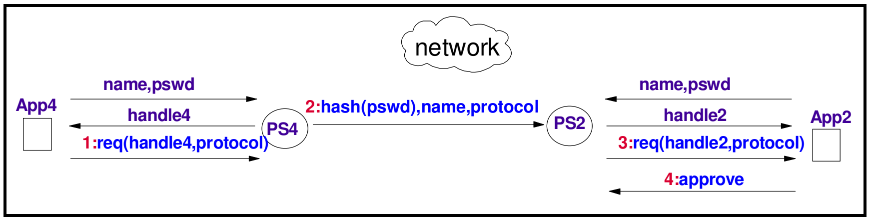

Figure 3: Password-based registration for distributed applications

1. **Password based:** Each client must register at its Pserver with the same (name,pswd).
   Client_i : sends *Registration_Req(name, pswd)* to *Pserver_i*
   Pserver_i : returns a handle *api_handle_i* to its client
   A request is serviced only if requested by a majority of servers. For that, if *Pserver_i* gets a request *Request_protocol(api_handle_i, protocol)* from *Client_i,* then *Pserver_i* sends (hash(pswd), name, protocol) to all. If at some *Pserver_j* no such request has arrived, it asks for approval from its *client_j* (after checking hash(pswd)).

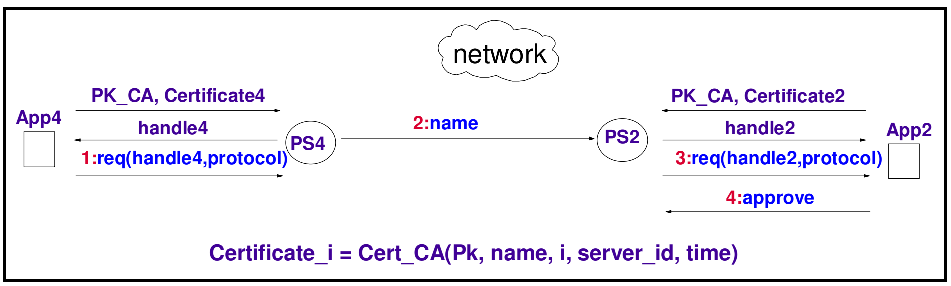

Figure 4: Certificate-based registration for distributed applications

2. **Certificate based**: Assume a Certificate Authority common to all clients. Prior to registration at the Pserver, each Client_i gets a one-time certificate
   Certificate_i = Cert_CA(Client_PubKey, name, i, server_id, time).
   This certificate is used to convince Pserver_i that the registration request indeed comes from a valid client which is the i_th component of some application named *name* (that is, if the same *name* is used at various clients, then they are instantiations of the same application). Of course, the application must also prove ownership of the secret key corresponding to the public key in the certificate (by signing time or a challenge from the proactive server). This protocol is depicted in Figure 4.
   The password solution is much weaker since if one finds out the password it can "forge" a client at all other machines at once, whereas a different certificate (and secret key) is needed at each machine.

5.2 Data Storage API's

These APIs provide *secure store and recovery* functionalities for the application data which are all based on the various Secret Sharing algorithmic techniques. The API supports the following functions:

StoreData (DataID, DataValue, data_type)
Retrieve (DataID)
Write (DataID, newValue)

where *DataID* uniquely identifies the data entry, and *data_type* indicates what security options/requirements apply to this data. Below we categorize the various *types* of data; specifically, data can be either read/write or write-once memory, can be public or private, distributed or local. Table 1 summarizes these types.

- **Public (non-secret)** data can be requested to be stored and retrieved either as a write-once data or with read/write privileges. The latter case makes no sense in a local (central) application, since then it could be erased by a single corrupted server while under attack. Hence, in order to change the secret value (namely, perform a write operation), a quorum of the Pservers must request the operation.
- **Secret write-once** data can either be "single writer/reader", hence it is private to its owner but is distributed (via secret sharing) among servers for security and tolerance, or "single writer/distributed reader" data which has already been distributed among the servers elsewhere and can be read/reconstructed only if a quorum of the servers request to read it. Note that the system automatically provides refresh and recovery of shares for this type of data.
- **Secret read/write** is "distributed writer/distributed reader". It requires a quorum of the servers to change its value, hence it is applicable only in the distributed scenario.

Read/Write Data	must accept request for store/retrieve from a quorum of servers	Same as write-once, but the secret value may be changed if requested by a quorum of servers	N.A.
Write Once Data	accepts request for store/retrieve from one server	"single writer/distributed reader". Receive secret shares that are computed elsewhere (by the app). Perform periodical refresh and share recovery when needed	"single writer/reader" Secret sharing w/dealer. Only owner may reconstruct. Can not be modified. Perform periodical refresh and share recovery when needed.
	Public (not secret)	**distributed**	**private**
		Secret Data	

An additional interesting secure storage service is *StoreUntil (DataID, Data Value, Date).* This is a special interesting service that is derived from the "single writer/distributed reader" or "distributed writer/distributed reader" variant. The data is kept secret until the specified date.

5.3 Service API's

The following services are readily available from the proactive network:

- *GenerateDSSKey (params)* - Pservers engage in the generation of a distributed DSS key
- *GenerateDSASignature(message, DSSKey)* - Pservers engage in the generation of a distributed signature, using the algorithm of [13].
- *GetTime()* - returns a vector with the local time at each proactive server.
- *GetRandom()* - every Pserver generates and sends a random number, return XOR on them.
- *GenerateJointSecret()* - engage in the protocol that generates a joint secret. Secret value can be either specified or a random value.

5.4 Communication API's

The proactive toolkit can supply means for proactively secure communication between two nodes, either by supplying proactive keys, or by supplying proactive communication-links.

Proactive Keys (Fig. 5) - Communication applications often require generation of session keys, and these keys need to be refreshed from time to time. Such refreshed keys can be generated by the proactive toolkit and supplied to the application, together with certificate (signed by the proactive key of the Pservers) which certifies the public key of period *t* (K_t).

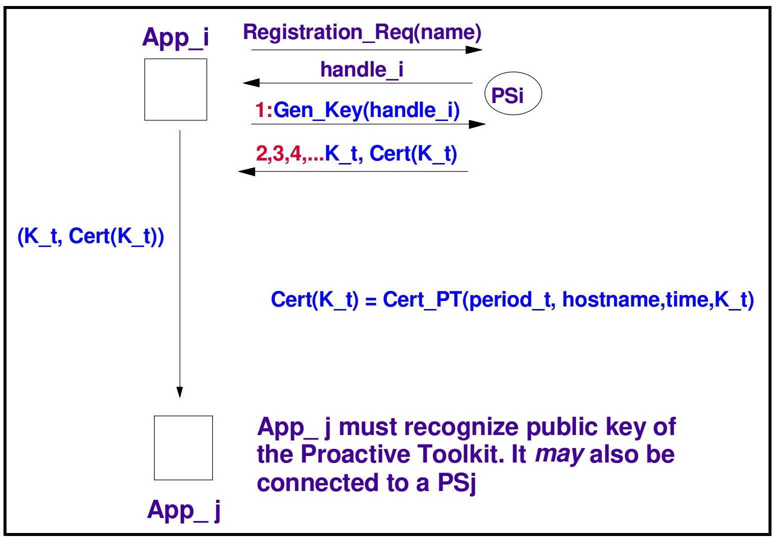

Figure 5 - Providing certified public keys each period

Proactively secure communication links (Figure 6) - Since the Pservers already maintain proactively secured communication among themselves, this mechanism can be provided as a service to an external application. To achieve that, an application first needs to register as a client on both servers (using the same name) and then use the send/receive API's between the client and the Pservers as shown in Figure

6 below.

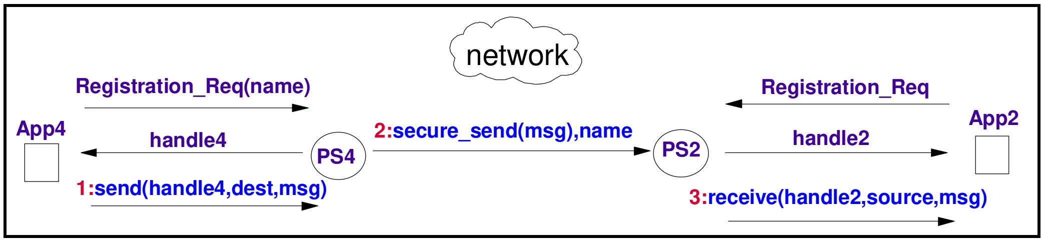

Figure 6 - Using the proactively secured links to send/receive messages.

6. Implementation

The Proactive Security Toolkit has been prototyped in Java 1.1. A beta version of the toolkit will be available in July 99 for public experimentation, and a running demo is in our web site.

6.1 Java Related Implementation Issues

The proactive environment architecture and its algorithms constitute quite a complex system to implement and test. As such, the Java language was a natural choice for implementation since it provides a fast and simple prototyping environment. Moreover, its portability across platforms was an important feature, since different nodes in the proactive network may have to run the toolkit on entirely different platforms (for example, our demo runs on a network of five nodes, some of which are UNIX based and others are Windows based). Finally, as we show in Section 6.3, it also provided simple means for remote Gui (i.e. via applets), so that management (system administration, testing etc.) of the entire proactive network can be conducted from a single point. Yet, this choice of programming lagnguage had a number of implications.

1. Erasing information from memory, which is an absolutely necessity for the correct implementation of secrets refresh, is an issue in all environments (due to virtual memory) and, in particular, in a garbage collected environment like Java, since garbage collectors typically copy memory as part of the collection proccess.
2. The proactive model assumes that after the adversary has lost control on a machine the code of the proactive server program is valid, so it is either protected by some tampered-proof memory device or can be safely loaded by the operating system (see the API section). The code for a Java program includes the code for the JVM (Java Virtual Machine), as well as the byte-code of all classes loaded (dynamically) by the machine in the course of its execution. Therefore, satisfying the code-validation assumption for Java programs may require assistance not only from the Operating System, but also from the JVM, possibly by using mechanisms like signed classes or by writing a customized class loader.
3. We were able to use some of the more advanced features of Java to simplify both the protocols and communication (by using serialization), and the API (by using dynamic class loading and reflection). All protocols and messages are implemented as subclasses of an abstract superclass. In this way all protocols are treated in a uniform way, which simplifies both the dispatch of messages to protocols and the addition of new protocols. In addition, we didn't have to define 'protocol messages' in a strict, well structured, way and parse them. Instead, all messages are sent as serialization of some object.

6.2 Signing an agreed-upon object

The following implementation issue, not necessarily exclusive to Java , is relevant in order to jointly sign and validate an agreed-upon object. An object is signed by first converting it to a number, but a conventional conversion may not guarantee that identical objects will be converted identically. For example, if two identical sets are implemented via linked lists then the representations may be different due to distinct orders within the lists.

In our implementation, it is often desirable that all "good" servers sign an identical object which they all posses. For that, the following protocol has been used:

- Each server *broadcasts* a byte array which is a serialization of the object
- A server accepts the byte array which is the serialization of an identical object to its own , and comes from the lowest indexed server.

6.3 API Implementation

Using the Java language enabled us to implement the API between the server and its clients in a convenient and simple way, similar to the API for writing applets. To write a proactive application the client must write a class which is a subclass of the class **ProactiveApplication** which is part of the toolkit. This superclass provides its subclass with methods to request services from the server, send messages to clients running on the other machines, and load new classes to the proactive server. In addition, this class defines abstract methods which the subclass must implement and which the server uses to notify the client about the status of request and about incoming messages from other clients. The class also defines an interface which the client implements to allow its data to be saved and restored from the server. In short, a large part of the API specifications outlined in Section 5 is already provided, either as methods, abstract methods or interfaces, of the **ProactiveApplication** superclass.

As a result, executing a "client'' application is essentially reduced to loading a class which is a subclass of ProactiveApplication into the JVM executing the Pserver class, and the issue of registering the client application is now reduced to an authorization/policy mechanism to allow the loading of this class. To this end, we suggest that the code for some predefined list of classes is part of the initial constants *C* of the server, and, as mentioned above, these classes can invoke methods to load new classes (in the same package or subpackages - to avoid namespace conflicts). A class is loaded at the next periodic refresh if a majority of the servers received a request from a client to load it. The code for all loaded classes is also validated at each refresh. Therefore, the server trusts classes that have been loaded to it, and the classes that are loaded initially are responsible for implementing the policy of which new client classes to load.

One natural policy that can be employed is via "signed code" mechanisms provided by the Java language. We intend to supply standard initial classes which provide a GUI interface to request loading of new classes, and which will only load classes that are signed by a predetermined certificate authority. However, different initial classes can implement different policies such as a "class that will delay requests for loading new classes for a week, while notifying managers and requesting their authorization". Some initial classes may decided not to load classes at all, but listen for requests through a TCP socket and decide whether to accept them on a per request basis (such a class can act as a proxy for a non-Java or a non-local client application).

Since there are no inter-process or inter-computer communication between the client and the server, the issue of authentication is much simplified. A misbehaving clients can bring down the server by exhausting resources but can not (modulo Java security) learn of other clients' data.

6.3 The ServerGui component

It is desirable to provide the ability to remotely inspect the proactive network, or any specific node within this network. To achieve this, we created the **ServerGui component**. This component is not part of the Pserver, but rather an independent program whose purpose is to send requests to a Pserver and to display the server's responses. It is written as a Java applet and can be downloaded throught the browser. The main request it supports is the "View PServer's Status and information", and as such it can function as a:

- *Gateway for the System Administrator*, which is an important role in a Proactively secured system. In the proactive model, protocols often end up by raising various types of alerts which must be resolved manually by a System Administrator with the assistance of log-files.
- *Debugging tool*. As a highly involved distributed system the proactive toolkit is difficult to debug, so information about the system's status, history and log files must be supplied.
- *Demo tool*. By extending the types of services requested by the ServerGui component, it is possible to demonstrate a rich set of mechanisms that the toolkit may support.

7. References

1. H. Attiya, and J. Welch, *Distributed Computing: fundamentals, simulations and advanced topics.* Mc.Grow-Hill, 1998.
2. G. R. Blakley, *Safeguarding cryptographic keys*. In Proc. AFIPS 1979 National Computer Conference, pp. 313-317. AFIPS, 1979.
3. D. Boneh and M. Franklin. *Efficient generation of shared RSA keys*. In Proc. Crypto '97, pp. 425-539.
4. R. Canetti, R. Gennaro, A. Herzberg and D. Naor, *Proactive Security: Long-term protection against break-ins.* CryptoBytes: the technical newsletter of RSA Labs, Vol. 3, number 1 - Spring, 1997.
5. R. Canetti, S. Halevi, and A. Herzberg. "*Maintaining authenticated communication in the presence of break-ins*". To be published in Journal of Cryptography, 1999. An extended abstract of this paper appeared in the Proceedings of the 16th ACM Symp. on Principles of Distributed Computation. 1997.
6. R. Canetti and A. Herzberg. *Maintaining security in the presence of transient faults.* In Crypto' 94, pp. 425-438, August, 1994.
7. CertCo, *Root Authority*, http://www.certco.com
8. C.S. Chow and A. Herzberg. *Network randomization protocol: A proactive pseudo-random generator.* Appears in Proc. 5th USENIX UNIX Security Symposium, Salt Lake City, Utah, June 1995, pp. 55-63.
9. P. Feldman. *A Practical Scheme for non-interactive verifiable secret sharing*. In Proc.28th Annual Symp. on Foundations of Computer Science, pp. 427-437. IEEE, 1987.
10. Y. Frankel, P. Gemmell, P. Mackenzie, and M. Yung. *Optimal resilience proactive public-key cryptosystems*. In Proc. 38th Annual Symp. on Foundations of Computer Science. IEEE, 1997.
11. Y. Frankel, P. Gemmell, P. Mackenzie, and M. Yung. *Proactive RSA*. In Proc. of Crypto '97.
12. P. Gemmell. *An introduction to threshold cryptography*. In Cryptobytes, Winter 97, pp. 7-12, 1997.

13. R. Gennaro, S. Jarecki, H. Krawczyk and T. Rabin, *Robust threshold DSS signature*. In Ueli Maurer, editor, Advances in Cryptology - Eurocrypt '96, pp. 354-371, 1996. Springer-Verlag Lecture Notes in Computer Science No. 1070.
14. V. Hamilton, G. Istrail - Sandia National Labs. *Implementation of proactive threshold public-key protocols*, Proceedings of the 1998 RSA Data Security Conference.
15. A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk and M. Yung. *Proactive public key and signature systems*, ACM Security '97.
16. A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, *Proactive secret sharing, or: How to cope with perpetual leakage*. In D. Coopersmith, editor, Advances in Cryptology - Crypto '95, pp. 339-352, 1995. Lecture Notes in Computer Science No. 963.
17. *ITTC* http://www.stanford.edu/~dabo/ITTC
18. A. Iyengar, R. Cahn, C. Jutla and J.A. Garay, *Design and implementation of a secure distributed data repository*, in IFIP 1998.
19. M. Malkin, T. Wu and D. Boneh, *Experimenting with shared generation of RSA keys*, in proceedings of the Internet Society's 1999 Symposium on Network and Distributed System Security (SNDSS), pp. 43-56.
20. M. Naor, B. Pinkas and O. Reingold, *Distributed pseudo-random functions and KDCs*, to appear in Proc. of Eurocrypt '99.
21. R. Ostrovsky and M. Yung, *How to withstand mobile virus attacks*, PODC 1991, pp.51-61.
22. T. Pedersen. *Non-interactive and information theoretic secure verifiable secret sharing.* In D. Davies, editor, Advances in Cryptology - Eurocrypto '91, pp. 522-526, 1991. Lecture Notes in Computer Science No. 547.
23. T. Pedersen. *A threshold cryptosystem without a trusted party.* in J. Feigenbaum, editor, Advances in Cryptology - Crypto '91, pp. 129-140, 1991. Lecture Notes in Computer Science No. 576
24. T. Rabin, *A simplified approach to threshold and proactive RSA*, Proc. of Crypto '98.
25. M. K. Reiter, *The Rampart toolkit for building high-integrity services*. In K. P. Birman, F. Mattern and A. Schiper, editors, *Theory and Practice in Distributed Systems* (LNCS 938), 99-110, Springer-Verlag, 1995.
26. M. K. Reiter, *Secure agreement protocols: Reliable and atomic group multicast in Rampart.* Proc. 2nd ACM Conference on Computer and Communication Security, 1994.
27. M. Reiter, M. Franklin, J. Lacy and R. Wright, *The Ω Key Management Service*, Proc. of the 3rd ACM Conference on Computer and Communication Security, 1996.
28. A. Shamir. *How to Share a Secret*. Communications of the ACM, 22:612-613, 1979.
29. T. Wu, M. Malkin and D. Boneh, *Building intrusion tolerant applications,* submitted to 8th USENIX Security Symposium.

A. Appendix - Review of Algorithms

A.1 Proactive Secret Sharing:

Secret Sharing (SS) protocols were introduced in [28,2] to protect the secrecy of a value by distributing it over several servers. Typically the secret is shared among n servers and any $t+1$ out of them can recover the secret. The proactive security of secret sharing schemes is maintained by having the servers periodically refresh their shares of the secret. The refreshment protocol guarantees that the new shares are independent of the old shares, but that they still define the same secret.

In **Shamir's scheme** [28], the secret is a value s in the set of integers $[0..q-1]$ where q is a prime; the dealer (who is sharing the secret) generates t random numbers $a_1,...,a_t$ modulo q. Given the polynomial $f(X)=s + a_1 X + ... + a_t X^t$ the dealer gives to server i the share $s_i=f(i) \bmod q$. It's clear that any t

servers have no information about s while $t+1$ can reconstruct the value by polynomial interpolation. **Periodic refreshments** of the shares can be performed as follows [21,16]. Each server i chooses a random t-degree polynomial $f_i(X)$ such that $f_i(0)=0$. Server i then sends to server j the value $s_{ij}=f_i(j) \bmod q$. Server j then computes its new refreshed share s'_j as follows: $s'_j = s_j + s_{1j} + \ldots + s_{nj} \bmod q$ and erases its old share. The new shares s'_i lie on the polynomial $f'(X) = f(X) + f_1(X) + \ldots + f_n(X)$ which is still of degree t and whose free term is still s so the new shares define the same secret. **Recovery** of the share $s_r=f(r) \bmod q$ of server r is also possible as follows [16]. Each server i chooses a random t-degree polynomial $g_i(X)$ such that $g_i(r)=0$ (by choosing the free coefficient of $g_i(X)$ to satisfy this condition). Server i then sends to server j the value $g_{ij}=g_i(j) \bmod q$. Server j then computes its new share g'_j: $g'_j = g_{1j} + \ldots + g_{nj} \bmod p$. The new shares g'_i lie on the polynomial $g'(X) = g_1(X) + \ldots + g_n(X)$ which is still of degree t and satisfies $g'(r) = 0$. Now, each server i sends to server r the value $s_i+g'_i \bmod q = f(i)+g'_i \bmod q$. Server r interpolates these values to construct the polynomial $f(X) + g'(X)$ from which it derives $f(r)+g'(r) \bmod q$ and this is exactly $s_r=f(r) \bmod q$, its lost share.

The above procedures work only in the case of a passive adversary who may read the data but not modify it or cause the behavior of a server to change. In case of an active adversary the above techniques were extended either by using **error-correcting** (EC) polynomial interpolation techniques [ber-welch] to tolerate erroneous values, or by using **Verifiable Secret Sharing** (VSS) protocols (see [12]). In particular, the VSS protocol by Feldman [9] proved especially suitable for this purpose and is used to detect malicious dealers, namely servers which deviate from the basic Shamir SS protocol and deal wrong values among the servers. Feldman's VSS protocol extends Shamir's SS protocol as follows. Given the polynomial $f(X)=a_0+ a_1 X + \ldots + a_t X^t$ the dealer sends to server i the share $s_i=f(i) \bmod q$ but also broadcasts to all servers the set of values $(\alpha_1, \ldots, \alpha_n)$ where $\alpha_i=g^{ai} \bmod p$. This *alpha-vector* allows the servers to check that the values s_i indeed define a secret by checking that $g^{si} = \Pi_j (\alpha_j^{\,i^j}) \bmod p$ (the numbers p,q and g satisfy some specific properties).

A.2 Joint Secret Sharing:

Joint secret sharing [22,23] is a scheme that allows the servers to *collectively* generate shares corresponding to a secret among n servers *without* the need of a dealer (like the secret-sharing schemes) so that any $t+1$ out of them can recover the secret. It calls for n secret-sharing protocols to occur simultaneously, where each server acts as a dealer of a secret s_i and the value of the resulting secret is $S = s_1+\ldots+s_n$. In the **Joint Random SS** each server chooses a *random* secret s_i so that S is also random. The **Joint Zero SS** asks that $si=0$ hence the resulting S remains 0. In both cases, each server simply sums the shares it receives from all other servers to obtain its own share of S. Clearly, one can categorize the Joint Secret scheme based on the various types of adversaries by using different types of secret-sharing schemes: Shamir's SS or the Verifiable SS.

A.3 Proactive Signature Scheme:

A proactive signature scheme involves three phases: the *key generation* phase (preferably done without a trusted dealer), the *joint signature-generation* phase and a periodical *proactive key refreshment* phase of the servers' key shares. The signature is generated in a *distributed* fashion from the shares of the key (the signature key is never reconstructed). Moreover, it has to hold that despite proactivization of the signing key the signature on a message m, computed under any of the representations of the key, is the same. Proactive solutions for few signature schemes have been devised, among them are distributed RSA signatures [3,11,24] and distributed DSS (Digital Signature Standard) signatures

[15,13]. Our toolkit implements the DSS solution, which is based on the threshold signature scheme of [13] combined with [15] and is outlined below.

First, recall the definition of the **Digital Signature Standard (DSS),** the adopted US standard for signature scheme.

Key Generation. Let p be a prime number of a specified length, q be a prime divisor of $p-1$ and g be an element of order q in Z^*_p. The triple (p,q,g) is public. The DSS key consists of a secret key x, which is a random number satisfying $1<x>q$, and a verification public key y satisfying $y = g^x \bmod p$.

Signature Algorithm. Input: m, the hashed message. Signature: pick a random number k between 1 and q, calculate $k^{-1} \bmod q$ and set $r = (g^{k^{-1}} \bmod p) \bmod q$, $s = k(m + xr) \bmod q$. Output: the signature on m is the pair (r,s).

Verification Algorithm. Verify that $r = (g^{ms^{-1}}\, y^{rs^{-1}} \bmod p) \bmod q$

The Proactive DSS algorithm is now:

Proactive DSS Key Generation. Jointly generate a DSS key by using the Joint Random Secret Sharing protocol to share a random value x. x is now the signing (private) key, and its corresponding verification key is g^x.

DSS Signature Generation. [13]

(i) Generate a Joint Random secret k

(ii) Compute the shares of $r = (g^{k^{-1}} \bmod p) \bmod q$; this is done via the protocol for computing the reciprocal of the joint secret k .

(iii) Use a protocol that computes the shares of a product of two secrets to compute the shares of the value $s = k(m+xr)$. Note that since m and r are known to all parties, and x is a shared secret, the shares of the value $(m+xr)$ are readily known to each server.

(iv) Each server outputs the pair (r,s_i).

The shares (r,s_i) are then combined (by a public procedure) to produce the signature (r,s).

Proactive Key-Refresh. Use the refreshment scheme of a secret to refresh the key x.

A.4 Agreement/Broadcast Protocol:

Some of the above protocols require that if a message needs to be broadcast to all other parties then "all good servers eventually receive the same message". This problem, a.k.a. the "agreement problem", has been studied extensively (see, for example, [1], [26]). Our model is a simplified model since we assume an upper limit communication delay between two points and, in addition, we assume that communication is authenticated (existence of signatures). The details of the protocol we have implemented will be described in the final version of the paper.