Secure Distributed Storage and Retrieval[*]

Juan A. Garay[†] Rosario Gennaro[‡] Charanjit Jutla[‡] Tal Rabin[‡]

Abstract

In his well-known Information Dispersal Algorithm paper, Rabin showed a way to distribute information in n pieces among n servers in such a way that recovery of the information is possible in the presence of up to t inactive servers. An enhanced mechanism to enable construction in the presence of malicious faults, which can intentionally modify their pieces of the information, was later presented by Krawczyk. Yet, these methods assume that the malicious faults occur only at reconstruction time.

In this paper we address the more general problem of secure storage and retrieval of information (SSRI), and guarantee that also the process of storing the information is correct even when some of the servers fail. Our protocols achieve this while maintaining the (asymptotical) space optimality of the above methods.

We also consider SSRI with the added requirement of *confidentiality*, by which no party except for the rightful owner of the information is able to learn anything about it. This is achieved through novel applications of cryptographic techniques, such as the distributed generation of receipts, distributed key management via threshold cryptography, and "blinding."

An interesting byproduct of our scheme is the construction of a secret sharing scheme with shorter shares size in the amortized sense. An immediate practical application of our work is a system for the secure deposit of sensitive data. We also extend SSRI to a "proactive" setting, where an adversary may corrupt all the servers during the lifetime of the system, but only a fraction during any given time interval.

Key Words: Information security, information dispersal, distributed storage, threshold cryptography.

[*]A preliminary version of this paper appeared in *Proc. 11th International Workshop on Distributed Algorithms (WDAG '97)* [18].

[†] Information Sciences Research Center, Bell Laboratories, 600 Mountain Ave, Murray Hill, NJ 07974. E-mail: `garay@research.bell-labs.com`. URL: `www.bell-labs.com/user/garay/`. Work done while the author was at the IBM T.J. Watson Research Center.

[‡] IBM T.J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598. E-mail: {`csjutla, rosario, talr`}`@watson.ibm.com`. URL: `www.research.ibm.com/security/`.

1 Introduction

The notion of *information dispersal* was introduced by Rabin [32] in his well-known Information Dispersal Algorithm (IDA). The basic approach taken in IDA is to distribute the information being stored, F, into n pieces among n active servers, in such a way that the retrieval of F is possible even in the presence of up to t failed (inactive) servers. The salient point was to achieve this goal while incurring a small overhead in needed memory. And indeed Rabin's result is space optimal. Retrieval of F is possible from $n - t$ pieces, where each piece is of length $\frac{|F|}{n-t}$. (For completeness, we include a short overview and example of IDA in Appendix A.)

In addition to its optimal space complexity, the IDA technique has very attractive properties as it permits any party in the system to retrieve the distributed information (by communicating with the piece holders); it does not require a central authority; it is symmetric with respect to all participants; and no secret cryptographic keys are involved. However, this combination of very desirable properties is achieved at the expense of limiting the kind of faults against which the algorithm is robust, namely, by assuming that available pieces are always unmodified.

An enhanced mechanism to reconstruct the information when more general faults occur was presented by Krawczyk [28], who called this problem–and its solution–the *Secure Information Dispersal* problem/algorithm (SIDA). This mechanism is able to tolerate malicious servers that can intentionally modify their pieces of the information, and is also space optimal (asymptotically). In a nutshell, SIDA makes use of a cryptographic tool called *distributed fingerprints*, which basically consists of each processor's piece being hashed—the *fingerprints*, and then distributing this value among all servers using the coding function of an error correcting code (e.g., Reed-Solomon [1]) that is able to reconstruct from altered pieces. This way, the correct servers are able to reconstruct the fingerprints using the code's decoding function, check whether pieces of the file were correctly returned, and finally reconstruct F from the correct pieces using the IDA algorithm.

Our contributions. A shortcoming of these methods is that they assume that the faults only occur at reconstruction time, after the dispersal of the pieces has been properly done. In this paper we address the more general problem of **secure storage and retrieval of information** (SSRI), and guarantee that also the process of storing the information is correct even when some of the servers fail. We consider the scenario in which a user interacts with the storage system by depositing a file and receiving a proof (in the form of a *receipt*) that the deposit was correctly executed.

For efficiency reasons our design makes the distributed nature of the system *transparent* to the user. This is achieved by having the client interact with a *single* server, called the *gateway* (GW). This design choice avoids the need for lengthy computations and, above all, parallel connections to several servers from the client. On the other hand, choosing the gateway option adds the extra technical difficulty of designing the protocol in a way that the gateway is not a single point of failure. (See Section 2 for further elaboration on the model.)

First we concern ourselves only with the **integrity** of the information, i.e. we require that retrieved data be correct. We introduce simple protocols that extend the above methods to enable storage in the presence of malicious faults, while maintaining the (asymptotical) space optimality of the above methods. Namely, each piece is of size $\frac{|F|}{n-t}$ plus a small quantity which does not depend on the size of the file (but on n and a security parameter). Our storage protocol is designed so that some form of consistency is maintained among the servers without incurring the cost of (potentially expensive) agreement protocols. Another important technical element of the storage protocol is the generation of receipts for the deposit of files through the application of distributed digital signatures. It is guaranteed by our protocols that a receipt is issued only when the correct information has been stored in the correct servers.

We also consider SSRI with the added requirement of **confidentiality** of the information being deposited, i.e., that any collusion of up to t servers (except ones including the rightful owner of the information) should not be able to learn anything about it. Confidentiality of information is easily achieved by encryption. Yet, this in turn poses the problem of *key management*, that is, the safe deposit—in the same storage system—of the cryptographic key used to encrypt the file that is being deposited. Under this scheme, how would the user be able to retrieve his file confidentially? Remember that in our design he communicates with the system through a single gateway, which means that if only the standard techniques of secret sharing reconstruction were used [35, 7], then the gateway would know all the information available to the user. One novel component of our confidentiality protocol for the solution of the above problem is its distributed key management aspect, achieved through the application of a combination of threshold cryptography (see Section 2.5) and blinding techniques [5].

The contributions of this paper can be summarized as follows:

- We consider the more general problem of information storage and retrieval, guaranteeing that also the process of storing the information is secure in the presence of (maliciously) failing servers. Our solutions have an (asymptotically) optimal blow-up factor, and tolerate up to $t < n/2$ malicious servers.

- Novel applications of cryptographic techniques, namely, the generation of receipts via distributed digital signatures, distributed key management via threshold cryptography, and blinding (together with threshold cryptography) in the context of decryptions rather than signatures.

- Secret sharing made "shorter:" An interesting by-product of our constructions is a (computational) secret sharing scheme which achieves shorter size shares, in the amortized sense, than the one of [29].

- "Proactive" SSRI: SSRI robust against an adversary which may corrupt all servers during the system's lifetime, but only up to t during each time interval [30, 3].

The remainder of the paper is organized as follows. In the next section we present the model, necessary definitions, and description of the tools that we use in this paper. In Section 3 we describe the protocols for basic SSRI (i.e., integrity only), while in Section 4 we present SSRI with the added requirement of confidentiality of the information. In Section 5 we present the scheme for secret sharing with shorter shares, while in Section 6 we show how to make SSRI "pro-actively" secure.

2 Model, Definitions, and System Considerations

In this section we describe our distributed model and give definitions for the task of secure storage and retrieval of information. We also list the cryptographic tools that we need in the sequel.

2.1 The distributed model

We start by describing an abstraction of the distributed system we consider. We consider a communication network with two classes of entities: the *users*, denoted $U_1, U_2, \cdots, U_m$, and the *servers*, denoted $V_1, \cdots, V_n$. We will sometimes refer to the servers collectively as V. It is among the servers that the distributed storage of the information takes place.

We model the communication among the servers by a completely connected graph of authenticated links; for the purpose of this paper we also assume a point-to-point communication link between each of the users and every server.[1] (Actually, only $t+1$ direct connections are needed, where t is an upper bound on the number of malfunctioning servers, as explained later.) Thus, links do not guarantee secrecy, but this can be achieved through the use of encryption. In fact, all the parties are assumed to be computationally bounded, so that the underlying cryptographic primitives that are used by our protocols (see Section 2.4) can be considered secure, and the security assertions that we make hold with high probability.

We assume the availability of a global clock, which allows the network computation to proceed as a series of *rounds*.[2]

It is assumed that at any time during the life of the system, at most t of the n servers can malfunction, possibly in malicious ways. Further, we assume that the faulty servers can even collude and act in concert in order to disrupt the computation—e.g., in a plain spoiling manner; try to prevent the storage or reconstruction of a file; or learn some information (e.g., a key) which the user wants to keep private. We also assume that $n > 2t$. The users, on the other hand, are assumed to always behave correctly; given the application, the case of malfunctioning users is not interesting and can be easily detected.

As mentioned in the Introduction, in our protocols the users will interact with a single, not necessarily the same, distinguished server called the *gateway* (GW). Nevertheless, our design shall be *uniform* in the sense that all servers will be able to perform the same distribution and recovery functions.

Remark 1 Interacting with a single server enables an adversary to create a simple "denial-of-service" attack by simply crashing the GW. However, as demonstrated in Section 3, our global clock (reliable time-out) and $n > 2t$ assumptions guarantee that a user will eventually contact a working server.

An alternative design choice would be to have the user contact each of the servers directly—call this the "multiple connections" model. The reasons for our choice of a single connection/single gateway are two-fold:

- Broad applicability: As already pointed out (footnote of Section 2.1), we aim at broad applicability, meaning that users (e.g., browsers) with not much of an add-on burden should be able to use the application we are proposing. This justifies the use of a single connection (e.g., `http`) at a time between the user and a gateway, as well as trying to minimize the number of functions (e.g., IDA; secret sharing; storage of many signatures) that have to be performed by the user.
- Efficiency considerations: Besides the functionality requirements, we would also like to minimize the computation and communication overhead on the user (in fact, in turn this also widens our applicability basis). As shown in Appendix D, both efforts are more demanding on the user in the case of multiple communications with the servers. (However, for a meaningful interpretation of the issues involved in the comparison, the reader should postpone its reading until after Sections 3 and 4.)

We now turn to the description of a major building block that we use in this paper.

[1] What we have in mind is Web implementations of our design. In such environments, authenticated communication can be realized through, e.g., `SSL` [27]. Similarly, point-to-point communication can be realized in various ways, and not necessarily through a direct connection.

[2] Again, this is for simplicity of exposition, as the only thing we need is a reliable time-out mechanism, and a means to guarantee the *freshness* of authentication. Possible realizations of the latter are via time stamps, or just *nonces*. See, e.g., [6].

2.2 Information Dispersal Algorithm

Rabin [32] proposed an algorithm that breaks a file F of length $L = |F|$ into n pieces F_i, $1 \leq i \leq n$, each of length $|F_i| = L/m$, so that every m pieces suffice for the reconstructing F. Note that the sum of the length of the pieces is $(n/m) \cdot L$. This algorithm, known as the *Information Dispersal Algorithm* has many applications to reliable storage and transmission of information. This algorithm is not only space efficient but also computationally efficient.

This algorithm should be contrasted with Shamir's algorithm [35] for secret sharing, which breaks a string F into n pieces each of the same size as F, so that F can be reconstructed from any m pieces. However, the main difference is that in this secret sharing scheme, any $m-1$ pieces give no information about F. On the other hand, in IDA less than m pieces may give some information about F.

In a later section (Section 5) we will show how IDA can be used to implement secret sharing more efficiently if we relax some of the conditions.

The actual details of the IDA algorithm (tuned to our application) are given in Appendix A.

2.3 Definitions

We now proceed to give the main definitions of our paper.

Definition 1 *An n-server system is a t-*resilient Secure Storage and Retrieval of Information system *(*SSRI *for short) if up to $t < n$ of the servers can malfunction, and for any user U holding file F there exist two protocols* `Deposit` *and* `Retrieval` *satisfying the following conditions:*

Deposit Availability: *User U wishing to deposit F will always manage to do so, and will receive a receipt (proof of deposit).*

Deposit Correctness: *If a receipt is generated by the servers for F, then each correct server has a copy of the file.*

Retrieval Availability: *User U will always be able to retrieve F.*

The above definition captures the notion of a storage system in which it is possible to store information, and which is able to preserve its integrity, even when a fraction of the servers malfunction; we will sometimes refer to such a system as "basic SSRI." The next definition extends the storage system to provide confidentiality as well.

Definition 2 *A* SSRI with confidentiality *is a* SSRI *system which additionally is t-private. Namely, the following condition also holds:*

Confidentiality: *No coalition of at most t parties (not including the rightful owner of the file) can learn anything about the contents of the file F.*

Note that the definition of Confidentiality also applies to coalitions which include the gateway. We now turn to a definition that measures the quality of information dispersal methods. The following paragraph and definition are taken from [28].

Reconstruction is possible in information dispersal methods because some redundancy is added to the n pieces into which the original information is partitioned. The amount of redundancy in an information dispersal method is typically measured by the following parameter.

Definition 3 *The* blow-up factor *(or just* blow-up*) of an information dispersal scheme is the ratio between the total size of the information being dispersed and the size of the original information. (By total size we mean the sum of sizes of all distributed pieces.)*

The blow-up of the original method of Rabin [32] is $\frac{n}{n-t}$, while the one of Krawczyk [28] is $\frac{n}{n-t} + o(1)$ (i.e., it requires an additional small quantity which does not depend on the size of the file). This is clearly (asymptotically) optimal if only $n - t$ pieces are to be used for reconstruction. Our methods also maintain this latter bound. We remark that reconstruction of information is also possible through error correcting codes. However, the inherent blow-up factor deteriorates to $\frac{n}{n-2t}$ in this case (see [1, 28]).

We now turn to describe the various cryptographic mechanisms that our protocols will make use of.

2.4 Cryptographic terminology and tools

The cryptographic primitives used in the protocols are summarized in Figure 1.

- **Keys:**

$\mathrm{VK}_U, \mathrm{SK}_U$	Public verification and private signing keys of user/party U.
$\mathrm{EK}_U, \mathrm{DK}_U$	Public encryption and private decryption keys of user/party U.
dk_{U,V_i}	Server V_i's share of private decryption key of user U.
CERT_U	Public key certificate of user U, which includes U, EK_U and CA's signature on EK_U.
$\mathrm{VK}_V, \mathrm{SK}_V$	Public verification and private signing keys of V.
sk_{V_i}	Server V_i's share of private key SK_V.

- **Cryptographic primitives:**

$\mathcal{H}(\cdot)$	A strong collision-resistant one-way hash function. Think of $\mathcal{H}(\cdot)$ as returning "random" values.
$\mathbf{E}_U$	Public key encryption using EK_U.
$\mathbf{S}_U(\cdot)$	Digital signature with respect to SK_U. We assume the signature function hashes the message before signing.
$\mathbf{S}_V(\cdot)$	(Distributed) digital signature with respect to keys $\mathrm{sk}_{V_1}, \cdots, \mathrm{sk}_{V_n}$.
$\sigma_{V_i}(\cdot)$	Partial digital signature with respect to sk_{V_i}
$\mathbf{e}_K$	Symmetric key-based encryption algorithm, taking key K and a plaintext, and producing the ciphertext.

Figure 1: *Keys and cryptographic primitives.*

All the users have two pairs of public/secret keys. (For simplicity, we will assume that the servers also act as the *certification authority* (CA), so that no third party needs to be involved in the transactions in order to verify the validity of the public keys.) One key pair $(\mathrm{SK}_U, \mathrm{VK}_U)$ is used for authentication ("signing") and verification purposes (resp.);[3] the other key pair $(\mathrm{DK}_U, \mathrm{EK}_U)$ is used for public-key decryption and encryption (resp.). DK_U is kept "shared" at the servers, each server V_i, $1 \le i \le n$, storing dk_{U,V_i}. Explanation of how this done and motivation are given in Sections 2.5 and 2.7, respectively. Similarly, the servers V share their own key pair $(\mathrm{VK}_V, \mathrm{SK}_V)$; that is, each server V_i stores sk_{V_i}, its share of private key SK_V. We make the usual security assumptions on these cryptographic primitives as summarized below.

We say that the signature scheme $\mathbf{S}$ is secure if it is *secure against adaptive chosen message attack* as defined in [23]. Informally, that means that an attacker who does not know the secret key SK and is given signatures on messages of its choice, will not be able to produce the signature for a new message.

We say that the encryption scheme $\mathbf{E}$, is secure if it is *semantically secure* as defined in [22]. Informally, that means that the encryption scheme is *randomized* and that the value $\mathbf{E}(m, r)$ (which denotes the public key encryption of message m with the help of pseudorandom number r) is computationally indistinguishable from a truly random string.

The following two subsections describe two major tools that we use in our protocols.

2.5 Threshold cryptography

The security of cryptographic protocols relies mainly on the security of the secret keys used in these protocols. Security means that these keys should be kept secret from unauthorized parties, but at the same time should always be available to the legitimate users.

Threshold cryptography is the name given to a body of techniques that help in achieving the above goals. In a nutshell suppose you have a key K which is used in the computation of some cryptographic function f on a message m, denote the result with $f_K(m)$. Examples of this include $f_K(m)$ to be a signature of m under key K, or a decryption of m under that key.

In a threshold cryptography scheme the key K is shared among a set of players $P_1, \ldots, P_n$ using a (t, n) *secret sharing* scheme [35]. Let K_i be the share given to player P_i. [4] Recall that by the definition of (t, n) secret sharing, we know that t shares give no information about K, but $t + 1$ shares allow reconstruction of the key K. The main goal of the threshold cryptography technique is to compute f_K without ever reconstructing the key K, but rather using it implicitly when the function f_K needs to be computed.

In the following we will use this terminology. Let the n servers $V_1, \ldots, V_n$ hold shares $\mathrm{sk}_1, \ldots, \mathrm{sk}_n$ respectively of a secret key DK which is the inverse of a public key EK.

A *distributed threshold decryption protocol* for $V_1, \ldots, V_n$ is a protocol that takes as input a ciphertext c which has been encrypted with EK (i.e., $c = \mathbf{E}_{\mathrm{EK}}(m)$ for some message m), and outputs m.

A *distributed threshold signature protocol* for $V_1, \ldots, V_n$ is a protocol that takes as input a message m and outputs a signature σ for m under SK.

The above protocols must be secure, i.e., they must reveal no information about the secret key DK, SK. A threshold cryptography protocol is called *t-robust* if it also tolerates t malicious faults.

Using threshold cryptography increases the secrecy of the key since now an attacker has to break into $t + 1$ servers in order to find out the value of K. Also, the basic approach increases the

[3]This is a natural assumption, as if for example the realization of our design is through a Web application, all browsers provide authentication in one form or another.

[4]There are two kinds of protocols for key generation: with or without a *dealer*. In a protocol with a dealer, it is assumed a trusted entity that produces the secret key K (with possibly an associated public-key K^{-1}), and then shares the key among the players. The dealer then "self-destroys." Notice that this assumes some trust in this entity since it knows the key in its entirety for a period of time. In a protocol without a dealer, the players themselves run a distributed protocol with some random inputs. This results in player P_i holding a share K_i of a secret key K.

availability of the key in the presence of so-called fail-stop faults (crashes); indeed, there is a need only for $t+1$ servers to be functioning in order to be able to compute the function F_K, meaning that one can tolerate up to $n-t-1$ crashes.

Threshold cryptography was originated in works by Desmedt [11], Boyd [2], Croft and Harris [8], and Desmedt and Frankel [12]. A survey of threshold cryptography techniques can be found in [13]. Protocols for discrete log-based threshold cryptosystems can be found in [2, 4, 12, 24, 31, 20]. Protocols for RSA-based threshold cryptosystems include [9, 10, 15, 19, 33]. In Appendix B we present an example of threshold cryptography applied to RSA [34].

The fault tolerance of the SSRI protocols we present in this paper ($n > 2t$) is inherited from the fault tolerance of the distributed threshold signature/decryption protocols [15, 19, 21, 14, 33], which is optimal.

2.6 Blinding

The cryptographic technique called *blinding* [5] can be explained as follows. Suppose that a server holds a secret key DK which allows it to compute decryptions in the public key encryption scheme $\mathbf{E}$. Assume also that the matching encryption key EK is known.

We say that the encryption scheme is *blindable* if the functions $\mathbf{E}_{\mathrm{EK}}$ and $\mathbf{E}_{\mathrm{DK}}$ are homomorphic, i.e., $\mathbf{E}(ab) = \mathbf{E}(a)\mathbf{E}(b)$.

Blindable encryption schemes allow to solve the following problem. A user wants to obtain the result of $m = \mathbf{E}_{\mathrm{DK}}(c)$ but without telling the server the value c he wants to be decrypted.

The user generates a random string r, computes the value $s = \mathbf{E}_{\mathrm{EK}}(r)$ using the public key EK and presents the server with the value cs which is random and thus gives no information about c. The server returns the value $\mathbf{E}_{\mathrm{DK}}(cs)$ which, by the homomorphic properties of $\mathbf{E}_{\mathrm{DK}}$, is equal to $\mathbf{E}_{\mathrm{DK}}(c)\mathbf{E}_{\mathrm{DK}}(s) = m \cdot r$. Thus, if the user divides the returned result by r he obtains the desired output.

Example of blindable encryption schemes include RSA [34] and most of the discrete-log based systems (e.g., ElGamal [17]). Appendix C contains an example of blinding using RSA.

A novelty of our scheme is the way we use blinding. Traditionally this technique was introduced to obtain signatures on messages that the server would not know [5]. This was in turn used to produce untraceable electronic cash. We use blinding in the context of decryptions rather than signatures in order to enhance the security of our distributed key management. The use of blinding will protect the privacy of the user's information against all servers (in particular the "gateway" GW—see Section 3), hence eliminating any single point of privacy failure from the system.

2.7 Key management

Our design takes advantage of the distributed environment to safely store some of the cryptographic keys used by the user. Recall from Section 2.4 that a user has two pairs of keys associated with him, one for authentication and the other for encryption. Regarding the storage of the private signing key SK_U, we have the user keep it (ideally in his smartcard, or, alternatively, managed by the application, e.g., browser). The reason is that there is no major security drawback in doing so, as if this key is compromised then the user can easily revoke it and get a new one. All past signatures generated under this key will still be valid, by the existence of the corresponding public verification key.

Conversely, the loss of the private decryption key DK_U would result in the loss of all the data encrypted under this key. For this reason, it is not advisable to store such a key in a single device, either held by the user or by the servers. Thus, in our design the private decryption key DK_U is kept shared at the servers, each server V_i, $1 \leq i \leq n$, storing dk_{U,V_i}. This way, more than t servers will have to be corrupted in order to recover DK_U. In order to decrypt a message for the user the servers will use such shares to run a distributed threshold decryption protocol. Further details about this procedure are given when we treat SSRI with Confidentiality in Section 4.

2.8 Comments on protocol presentation and optimization

Before we turn to the presentation of our protocols, we remark that various optimizations can be carried out. These include reducing the number of "echo" messages; reducing their size (e.g., only re-transmit the file when necessary, send its hash instead); sending acknowledgment messages upon receiving a file back; arranging fields so that cryptographic operations do not have to be computed twice; the use of "nonces," or transaction id's in order to prevent so-called "re-play" attacks; coping with "denial of service" attacks from incorrect servers, etc. In the presentation we omit such details for clarity's sake.

3 Integrity Only

The protocols of this section extend the methods of [32, 28] for integrity to achieve SSRI while maintaining (asymptotically) the space optimality. Namely, each piece of the file F deposited at each server is of size $\frac{|F|}{n-t}$ plus a small quantity which does not depend on the size of the file. We distinguish the following three transactions in basic SSRI:

- `Deposit`: User U contacts the gateway GW, deposits file F, and gets a receipt.
- `Dispersal`: The actual information dispersal takes place among the servers V_j.
- `Retrieval`: The user contacts GW to get F back.

We would like the protocols for `Deposit` and `Retrieval` to satisfy Definition 1. The protocol for `Dispersal` will ensure our claimed blow-up factor.

3.1 Deposit

The `Deposit` protocol is initiated by user U. The user contacts one of the servers (GW) and transmits a request for deposit (this request includes a digital signature on the file being stored). The user will conclude that the deposit has been carried out successfully once she receives a receipt from GW.

Figure 2 shows the (fault-free) flow of the protocol for `Deposit`.

We now describe the protocol in more detail. In DRequest, the user contacts GW and submits the file she wants to deposit, together with her signature on the file under her private signing key. In DExecution1 the GW forwards the request from the previous flow to the (remaining) servers. In DExecution2, every server receiving a valid message from GW (i.e., one whose signature verifies) "echoes" this message request to every other server; in the case of an invalid message, the server discards the request. Servers receiving at least one valid message store F as a valid request from user U. In DExecution3, each server V_i receiving the DExecution1 message from GW uses its share sk_{V_i} of the private signing key SK_V to generate a partial signature on F and U, and sends this message to GW. Servers not receiving a DExecution1 message from GW do not participate. In Receipt, the GW uses the partial signatures received from the other servers to compute the distributed digital signature on F and U, and sends it to the user, who verifies the signature using VK_V. This constitutes the receipt for the user's deposit. The user stores it for future use. Note that GW must have (at least) $(t+1)$ partial signatures in order to generate the receipt (cf. Section 2.5).

- **Fields:**

F	User's file to be deposited at the servers.

- **Protocol Flows:**

$$\begin{array}{llrcl}
\textsf{DRequest} & : & U & \xrightarrow{F, \mathbf{S}_U(F)} & \mathrm{GW} \\
\textsf{DExecution1} & : & \mathrm{GW} & \xrightarrow{F, \mathbf{S}_U(F)} & V_j,\ \forall j \\
\textsf{DExecution2} & : & V_j,\ \forall j & \xrightarrow{F, \mathbf{S}_U(F)} & V_i,\ \forall i \\
\textsf{DExecution3} & : & \mathrm{GW} & \xleftarrow{\sigma_{V_j}(U,F)} & V_j \text{ from } \textsf{DExecution1} \\
\textsf{Receipt} & : & U & \xleftarrow{\mathbf{S}_V(U,F)} & \mathrm{GW}
\end{array}$$

Figure 2: Sketch of the `Deposit` Protocol

If the user does not receive a receipt from GW after some pre-specified period of time, she tries a different server.

We now show the following about the `Deposit` protocol.

Lemma 1 *If the signature scheme* **S** *is secure, then protocol* `Deposit` *satisfies the* Deposit Availability *and* Correctness *conditions of Definition 1.*

Proof: Deposit Availability. If GW does not respond to the user with a receipt, then user U will turn to another server in order to deposit the file. As, by assumption, the number of servers $n > 2t$ and the design of the servers is uniform, it is guaranteed that the user will eventually contact a correct GW. (For example, if U chooses the next server at random, then the expected number of trials will be 2.) Once this happens every correct server V_i gets the user's message, verifies the message authenticity using the user's public verification key, and replies to GW with a partial signature under share sk_{V_i} of private key SK_V. GW combines the partial signatures and sends a correct receipt to the user. Because the signature scheme is secure a correct signature can be generated only by the user, thus the servers will not store files that were not produced by the user.

Deposit Correctness. Under the assumption that the signature scheme is secure, if a receipt is generated for file F then there were at least $t + 1$ partial signatures generated for this file under the server's shares of the private key SK_V. As by assumption at most t of the servers can be faulty, at least one of the partial signatures was generated by a correct server. A correct server generates a partial signature only if it has received a valid deposit request from GW, and in this case it also echoes the file to all the other servers (DExecution3 in Figure 2). Hence, every correct server has a copy of the file.

$\square$

- **Fields:**

F	User's file to be dispersed among servers $V_i, 1 \leq i \leq n$.
F_i	Portion of file F dispersed at server V_i.
$\mathcal{H}(F_i)$	Hash of F_i.

- **Protocol Steps:**

```
Each server $V_i$, $1 \leq i \leq n$, does:

    $\forall j$, $1 \leq j \leq n$, compute $F_j = F \cdot T_j$ (IDA);
    $\forall j$, $1 \leq j \leq n$, compute $\mathcal{H}(F_j)$;
    save $F_i$ and $\mathcal{H}(F_j)$, $1 \leq j \leq n$.
```

Figure 3: `Dispersal` Protocol.

3.2 Dispersal

There is no communication involved in the `Dispersal` transaction; it basically consists of a local computation at each server. `Dispersal` is initiated by the servers right after receiving a deposit request whose signature is verified, regardless of whether the request was received from GW directly, or as an echo from another server. Each server computes the pieces of F corresponding to all the servers using IDA, then computes the corresponding hashes of the pieces, and saves its own piece of the file and *all* the hashes. The sketch for this transaction is shown in Figure 3.

Lemma 2 *Protocol* `Dispersal` *achieves a* $\frac{n}{n-t} + o(1)$ *blow-up.*

Proof: Each server saves its own IDA portion of the file $|F_i|$, plus all the hashes $\mathcal{H}(F_j)$, $1 \leq j \leq n$. Since, for all i, j, $|F_i| = |F_j|$ and $|\mathcal{H}(F_i)| = |\mathcal{H}(F_j)| = O(1)$, for all i, j. the space required at each server is $|F_i| + n\ |\mathcal{H}(F_i)|$, and the total space is $n\ |F_i| + n^2\ |\mathcal{H}(F_i)|$. The first term follows from the IDA blow-up bound, and the second from the fact that $|\mathcal{H}(F_i)|$ is independent of the size of F. $\square$

In contrast, Krawczyk [28] suggests to share the hashes of the pieces themselves using Reed-Solomon codes. The space required by that method at each server is $|F_i| + \frac{n}{n-2t}\ |\mathcal{H}(F_i)|$. Thus, our approach is slightly less storage-efficient, but with the advantage of avoiding the complexity of the coding plus the communication. (Also note that for values of realistic implementations—e.g., $n = 5$ and $t = 2$—the storage requirements would be identical.)

3.3 Retrieval

`Retrieval` is the transaction initiated by a user in order to retrieve a file she has previously deposited, and for which she has received a receipt. Our protocol for `Retrieval` satisfies the Retrieval Availability condition of Definition 1. The protocol is shown in Figure 4.

We now describe the protocol in detail. In Retrieval Request the user contacts GW (not necessarily the one through which she deposited the file), and presents a signed request for the retrieval of the specific file. In RExecution1, GW sends this request to the other servers. Upon receiving the

- **Fields:**

F_j	Portion of file F stored in server V_j.
$\mathcal{H}(F_j)$	Hash of F_j.

- **Protocol Flows:**

RRequest : $U \xrightarrow{\quad \mathbf{S}_U(F) \quad} \mathrm{GW}$

RExecution1 : $\mathrm{GW} \xrightarrow{\quad \mathbf{S}_U(F) \quad} V_j,\ \forall j$

RExecution2 : $\mathrm{GW} \xleftarrow{\quad F_j,\ \mathcal{H}(F_i), 1 \leq i \leq n \quad} V_j,\ \forall j$

GW computes:

$\forall j,\ \mathcal{H}(F_j) \leftarrow$ majority of received $\mathcal{H}(F_j)$;

G : set of good indices; $G \leftarrow \emptyset$;

$\forall j,$ if F_j evaluates to $\mathcal{H}(F_j)$ then $G \leftarrow G \cup \{j\}$;

$F \leftarrow \sum_{i \in G} F_i \cdot T_i^{-1}$ (reconstruct with IDA)

Delivery : $U \xleftarrow{\quad F \quad} \mathrm{GW}$

Figure 4: `Retrieval` Protocol.

request the servers check the validity of the signature on the request and the ownership of the file. If the signature is valid and the user is the rightful owner of the file, then the server sends its piece of the file and the hashes of all the other servers' pieces to GW (RExecution1). As some of the servers might be faulty, they might send corrupted values to GW. GW establishes which hashes are the correct ones by computing majority, and discards those pieces whose hash does not evaluate to the computed one. Finally, the GW reconstructs the file using the remaining pieces using IDA, and sends the file to the user (Delivery). Upon receiving the message supposedly containing file F, the user verifies the authenticity of the file (by, for example, matching it against the receipt on the file which she had kept for control during `Deposit`). As before, if the user does not get a response from GW after some pre-specified period of time, or if she receives a file from the GW whose signature does not verify, she proceeds to contact another server.

Lemma 3 *Protocol* `Retrieval` *of Figure* 4 *satisfies the* Retrieval Availability *condition of Definition 1.*

Proof: By the same reasoning as in Lemma 1, we can assume that the user contacts a correct GW; then all the correct servers get the user's request. As we assume that the user has in fact previously deposited the file and received a receipt, we are guaranteed that each correct server has saved its piece of the file and hashes of all the pieces (this comes from the Deposit Correctness property). The servers then send their pieces and hashes of all pieces to GW. As some of the servers might be faulty, they might send corrupted values; however, the fact that $n > 2t$ allows GW to determine what hashes are correct (i.e., have not being altered) by applying majority. Finally, GW applies IDA to the correct pieces—those which evaluate to the correct hashes—to reconstruct the file. $\square$

Lemmas 1, 2 and 3 allow us to corroborate our claims of a basic SSRI system with an asymptotically optimal blow-up:

Theorem 1 *If* $\mathbf{S}$ *is a secure signature scheme, then protocols* `Deposit`*,* `Dispersal`*, and* `Retrieval` *of Figures 2, 3, and 4, respectively, satisfy Definition 1. Also, the blow-up of the stored information is asymptotically optimal.*

Basic SSRI provides a stable storage system, but does not give any guarantees about the secrecy of the information, as IDA does not provide it, and files flow in the clear between the users and the servers. The question of how to add confidentiality is treated in the next section.

4 Integrity plus Confidentiality

There might be a need in some applications to store files whose contents must remain private, i.e., that only the owner of the file would be able to read it. In this section we show how to provide confidentiality on top of the information integrity which basic SSRI provides, as specified by Definition 2.

A first solution that comes to mind (and which will be the building block of our final solution) is for the user to generate a symmetric key (e.g., a DES key) FK, encrypt the file F with FK, deposit the encrypted file with the servers and to store the key FK in its local memory. This simple solution satisfies the definition but has a major drawback. Now, not only the secrecy of the data relies on the encryption key FK, but also the availability of the data, as the loss of the key is equivalent to the loss of *all* the data encrypted with it. It is clear that simply storing the encryption key in the user's memory would be a very weak link in any construction, and hence we need to provide this key with storage security and availability which are comparable to the ones provided for the file itself. Thus, the natural solution is to store in some manner the encryption key itself in SSRI. However, note that simply storing the key in the clear would violate the confidentiality requirement.

4.1 Distributed key management and blinding

Thus, the question of SSRI with confidentiality can be reduced to the question of how to secretly store and retrieve the file encryption key FK, as the encrypted file can be treated as a regular file, which we already know how to deposit and retrieve. We now present our solution to the problem.

Recall from Section 2.7 that in our design each user U has a public/private key pair for encryption/decryption, $\mathrm{EK}_U, \mathrm{DK}_U$, where DK_U is kept shared among the servers using threshold cryptography, each server V_j storing dk_{U,V_j}. The user stores the symmetric encryption key FK by first computing $\mathbf{E}_{\mathrm{EK}_U}(\mathrm{FK})$, and then using the `Deposit` protocol of Figure 2 to deposit this value with the servers.[5] The fact that DK_U is kept shared at the servers does not allow any coalition

[5]In fact, for efficiency reasons which will be made clear later, a variant of the protocol should be used that keeps the encrypted key in its entirety at each server; as this is a small quantity, this variant does not pose a blow-up problem.

of up to t servers (even those including GW) to decrypt $\mathbf{E}_{\mathrm{EK}_U}(\mathrm{FK})$, and so far the Confidentiality requirement is satisfied.

However, upon a request from the user to retrieve the key and the file, if the servers used their shares of DK_U to send their partial decryptions of $\mathbf{E}_U(\mathrm{FK})$ to GW, GW would then be able to combine the partial decryptions and extract FK, in direct violation of the confidentiality requirement. To circumvent this problem we use "blinding" (see Section 2.6) in a novel way.

In a nutshell, the user will start the retrieval process by first generating a random integer r, which will be used as a *blinding* factor. He will then compute $\mathbf{E}_U(r)$ sign it and send it to GW, who will distribute it to the servers. Now, instead of each server V_j computing the partial decryption of $\mathbf{E}_U(\mathrm{FK})$ using dk_{U,V_j}, they will compute the partial decryption of the **product** $\mathbf{E}_U(\mathrm{FK})\cdot\mathbf{E}_U(r)$. As a consequence (assuming that $\mathbf{E}$ is blindable), GW will recover the product $r\cdot\mathrm{FK}$, which provides no knowledge about FK. On the other hand, the user will be able to retrieve FK by factoring out r. A detailed description of the blinding process is given in the next section. Note that this approach requires the user to store r securely, though temporarily.

One last issue which needs to be dealt with in this context is the following. Upon receiving the decrypted blinded key, how does the user know that this is the right value? (Note that this problem does not arise because of the blinding, but is in fact a general problem.) This will be solved by adding another round of communication among the servers in which every server V_i (not just GW) will reconstruct the blinded key, and then have the server sign this value under sk_{V_i}. Thus, the user will receive the blinded key signed under SK_V. It remains to be proved that in this case the user in fact receives the correct blinded key. This will be shown in the next section, where we describe our full protocol for SSRI with confidentiality and prove its correctness.

4.2 Deposit and retrieval with confidentiality

We will call the protocols for SSRI with confidentiality `Conf-Deposit`, `Conf-Dispersal`, and `Conf-Retrieval`. `Conf-Dispersal` will be identical to the protocol of Figure 3. `Conf-Deposit` is a slight modification of protocol `Deposit` in Figure 2. The user generates a symmetric key FK, computes $\bar{F} \overset{\triangle}{=} \mathbf{e}_{\mathrm{FK}}(F)$, and uses protocol `Deposit` to store the encrypted file $\bar{F}$ at the servers. Additionally, the user encrypts the key FK using his public encryption key EK_U (i.e., $\mathbf{E}_{\mathrm{EK}_U}(\mathrm{FK})$) and also sends this value to GW, who distributes it to all the other servers (with the echo step of Figure 2). The servers then apply `Conf-Dispersal` to disperse $\bar{F}$ but keep the encrypted key as is. (The reason for this is that as the encrypted key is a relatively small quantity, the IDA dispersal process will typically not be worth the effort.) As in the case of basic SSRI, the user receives a receipt, $\mathbf{S}_V(U,\bar{F},\mathbf{E}_{\mathrm{EK}_U}(\mathrm{FK}))$, for the successful deposit of the pair (encrypted file, encrypted key). Lemma 1 applies to `Conf-Deposit` straightforwardly.

The `Conf-Retrieval` protocol is shown in Figure 5. It assumes a blindable encryption scheme $\mathbf{E}$. At retrieval time, the user U generates a retrieval request by generating a random integer r, the blinding factor. He then computes $b=\mathbf{E}_U(r)$; he signs b and the identifier of the file he is trying to retrieve (say, the hash of the encrypted file) using his signing key SK_U, and sends the whole thing to the GW (CRequest). He also stores r securely. GW forwards this request to all the servers. (CExecution1). Each server V_j checks that the user signing this request has permission to access the file and the encrypted key, and if so, generates a partial decryption P_j of the blinded key using dk_{U,V_j}, its share of the user's key DK_U. That is each server produces a partial decryption P_j of the value $b\cdot\mathbf{E}_U(\mathrm{FK})=\mathbf{E}_U(r\cdot\mathrm{FK})$. The servers then distribute their partial decryptions to all other servers (CExecution2). Having enough partial decryptions from the other servers, the servers are able to compute $P=\mathrm{FK}\cdot r$; they also generate a partial signature on this value. In CExecution3, they send to GW their pieces of the encrypted file and all the hashes, together with their partial

- **Fields:**

r	Blinding factor: random number chosen by user U.
b	$\mathbf{E}_U(r)$.
$\bar{F}$	Encrypted user file stored in V.
$\bar{F}_j$	Portion of the encrypted file dispersed at server V_j.
P_j	Partial decryption using dk_{U,V_j} of $\mathbf{E}_U(\mathrm{FK} \cdot r)$.
P	$P = \mathrm{FK} \cdot r$.

- **Protocol Flows:**

CRequest : $U \xrightarrow{b, \mathbf{S}_U(b, \bar{F})} \mathrm{GW}$

CExecution1 : $\mathrm{GW} \xrightarrow{b, \mathbf{S}_U(b, \bar{F})} V_j,\ \forall j$

CExecution2 : $V_j,\ \forall j \xrightarrow{P_j} V_j,\ \forall j$

CExecution3 : $\mathrm{GW} \xleftarrow{\bar{F}_j,\ \mathcal{H}(\bar{F}_i), 1 \leq i \leq n,\ \sigma_{V_j}(P)} V_j,\ \forall j$

GW computes $\bar{F}$ as in Figure 4;

also computes $\mathbf{S}_V(P)$.

Delivery : $U \xleftarrow{\bar{F}, P, \mathbf{S}_V(P)} \mathrm{GW}$

Figure 5: Protocol `Conf-Retrieval`: Retrieval with confidentiality.

signatures on the blinded key. As in the original retrieval protocol, GW reconstructs $\bar{F}$ using IDA, and sends $\bar{F}$ and the signed blinded key back to the user (Delivery). The user obtains the file key FK by dividing out r, and decrypts the file. We now argue the correctness of the protocols.

Theorem 2 *If* $\mathbf{S}$ *is a secure signature scheme and* $\mathbf{E}$ *is a secure blindable encryption scheme, then protocols* `Conf-Deposit`, `Conf-Dispersal`, *and* `Conf-Retrieval` *satisfy Definition 2. Also, the blow-up of the stored information is asymptotically optimal.*

Proof: The proof of Deposit Availability and Correctness and of the bound on the space blow-up is similar to that of Lemma 2.
Confidentiality. In order to prove that the scheme is t-private, i.e., that no subset of size at most t has any knowledge of the content of the file, we need to show that no information about FK is ever known to the servers. Because $\mathbf{E}$ is a semantically secure encryption scheme, any information about the key FK can be computed only by decrypting it using the key DK_U. Because we assume that the threshold decryption protocol used by the servers is t-secure, no subset of t or less servers can

thus decrypt the key or any partial information about it. So the only value a coalition of t or less servers sees is the blinded key $P = r \cdot \mathrm{FK}$. Due to the blinding properties (r is a randomly-chosen number), the value P is a randomly distributed value which gives no information about FK.
Retrieval Availability. The argument of Lemma 3 also applies here. However, due to the encryption/decryption process, it must be additionally shown that the user receives the right value for the decryption key FK. The signed blinded key that the user receives is the correct one, due to the following. A correct GW forwards the request for decryption of $\mathbf{E}_U(\mathrm{FK})$ to all servers. As the request is a proper request signed by the user, the correct servers decrypt the value $\mathbf{E}_U(\mathrm{FK}) \cdot b$, that is, the encrypted key multiplied by the blinding factor. Thus, each correct server computes the correct blinded key, and generates a partial signature on it. As GW can only generate signatures with the participation of the correct servers, the signature will be on the valid blinded key. Finally, the user holds the value r, so he is able to compute FK by dividing r out. $\square$

5 Secret Sharing Made Shorter

An application of our result which is interesting in its own is an improvement on the size of the shares for computationally-secure secret sharing protocols [29]. Recall that in a (threshold) secret sharing protocol a dealer shares a secret s among n servers so that t servers cannot reconstruct it, but $t + 1$ can.

It is a well known fact that for an information theoretically-secure secret sharing protocol (i.e., one in which t shares give no information about the secret even when infinite computing time is given), the size of the shares must be at least the size of the secret. In [29] Krawczyk shows that if one relaxes the notion to one of "computationally secure," then it is possible to obtain shares of size $\frac{|s|}{t+1} + \ell$ where ℓ depends only on a security parameter. His idea goes as follows:

- Choose a key K for a symmetric encryption scheme $\mathbf{e}$ of length ℓ.
- Encrypt the secret to be shared; let $\sigma = \mathbf{e}_K(s)$.
- Use IDA [32] to distribute σ among the servers so that $t + 1$ pieces are enough to reconstruct σ; let σ_i be the piece given to the i^{th} server.
- Share K with an information theoretically secure scheme as in [35]; let K_i be the share given to the i^{th} server.

By the IDA bound we know that $|\sigma_i| = \frac{|\sigma|}{t+1}$. Clearly $|\sigma| = |s|$ and $|K_i| = \ell$, hence the stated bound.

Our SSRI protocol with confidentiality of Section 4 can be thought as a computationally secure secret sharing scheme. In it we have the servers sharing a secret key SK_V for an asymmetric encryption function $\mathbf{E}$. Let sk_{V_i} be the share of SK_V held by the i^{th} server. The user who deposits a file can be thought of as a dealer sharing a secret s according to the following steps:

- Choose a key K of length ℓ for a symmetric encryption scheme $\mathbf{e}$.
- Encrypt the secret s; let $\sigma = \mathbf{e}_K(s)$. Encrypt the key with the public key EK_V of the servers; let $\tau = \mathbf{E}_V(K)$.
- Use IDA [32] to disperse σ and τ among the servers so that $t+1$ pieces are enough to reconstruct σ; let σ_i and τ_i be the pieces given to the i^{th} server.

Let m be the length of the keys used by $\mathbf{E}$, i.e., $m = |\mathrm{SK}_V|$. Typically we have $m > \ell$. We can assume that $|\tau| = m$, thus each server keeps only a share of size $\frac{|s|+m}{t+1}$ for each secret s, plus the server holds the share sk_{V_i} (which is of size m), but that can be used for several sharings.

Now let's compare the asymptotic space requirements when sharing N secrets. In the scheme of [29] the storage requirement is clearly $N(\frac{|s|}{t+1} + \ell)$. In our scheme the storage needed is $N(\frac{|s|+m}{t+1}) + m$. So for large N (i.e., when $N\ell > m$) our scheme requires less storage.

6 Proactive SSRI

The protocols described in the previous sections withstand the presence of an adversary that can read the memory and corrupt the behavior of at most t servers during the whole lifetime of the system.

If such lifetime is long then the assumption that only t servers can be broken into may become unreasonable or too optimistic. *Proactive Security* [30, 3] is an area of research that deals with secure distributed systems in the presence of an adversary that may corrupt *all* the servers during the whole lifetime of the system, although only t at a time (i.e., the assumption is that during a pre-specified interval of time, say a day, the adversary may break into at most t servers).

Several proactive techniques have been presented in the past. Proactive protocols for secret sharing were presented in [26], while proactive protocols for threshold cryptography were introduced in [25, 20, 16].

A basic technique of Proactive Security is to introduce *refreshment* phases in the system. During a refreshment phase a server that has been broken into but is not anymore under the control of the adversary, can be restored to its initial state. In particular, all the data destroyed or modified by the adversary is restored with the help of the other servers. Also all secret information (e.g., cryptographic keys) contained in all the servers is somehow randomized so that the information leaked to the adversary in the previous time intervals will be useless in the future. Refreshment phases are invoked periodically regardless of the fact that break-ins have been detected or not.

The "proactivization" of our distributed storage system poses several interesting questions. At refreshing time we need to restore the memory of potentially compromised servers. This can indeed be done as by assumption only a minority of the servers might have been broken into during the previous interval. However, such a restoring operation can be potentially very expensive. Indeed, in order to restore the pieces of a server we need to recompute all the files and disperse them again. This means that at refreshing time the whole memory of the system has to circulate around in order to restore eventual break-ins. This can potentially be an enormous task and should be performed only if strictly necessary. For example, if in the previous interval the adversary did not produce any damage (or corrupted only a small fraction of the memory of the system), the above task would be too expensive.

What we need is a form of "adaptive" proactiveness in which the system performs the expensive restoring only when it is really necessary, while routine refreshment phases are cheaper to perform.

We describe our solutions, first for the integrity-only case (which is really the most interesting and novel) and then for the integrity plus confidentiality (which is just an application of proactive threshold cryptography).

Integrity Only. Recall from Section 3 that each file being deposited is first dispersed using our variation of SIDA [28]. This means that each server V_i, $1 \leq i \leq n$, will have an IDA piece of F, F_i, plus all the "fingerprints" of all the pieces $\mathcal{H}(F_1), \ldots, \mathcal{H}(F_n)$. By assumption during any given time interval only a minority of the servers can be corrupted. At the beginning of the refreshing phase each server broadcasts to the other servers the fingerprints. Server V_i takes a majority vote among the received fingerprints to identify the correct ones. It then checks if its own fingerprints are correct. If they are corrupted, it replaces them with the correct ones. It then checks its own IDA piece F_i against the correct fingerprint $\mathcal{H}(F_i)$. If the piece has been modified it broadcasts a message asking the other servers to reconstruct F_i for it. It then takes majority among the received responses to identify the correct F_i.

Notice that if the adversary was not present (or did no damage) in the previous time interval, then only the fingerprints of the stored files must circulate during the refreshment phase. This is clearly a negligible communication cost compared to the whole storage of the system. If the adversary did some damage, then the communication complexity of the refreshing phase is still proportional only to the amount of information the adversary corrupted and not to the whole memory of the system.

Integrity and Confidentiality. In this case the refreshment phase will consist first of all of the integrity-only refreshment phase, carried out on the encrypted files. However, in this scenario we need to worry about an adversary who besides corrupting the files, might also read the shares of the users' secret keys kept at a server. Once he reads more than $t + 1$ of such shares the adversary will be able to decrypt the users' files. But the shares of the secret keys can be proactivized using the proactive techniques used in threshold cryptography (for discrete log-based schemes see [25]; for RSA-based schemes see [16]). The refreshment phases for proactive threshold cryptography schemes have a communication complexity proportional to the size of the keys. So once again in the optimistic case (i.e., when the adversary does not corrupt the memory of the system) the work done in a refreshment phase is very small compared to the potential amount of memory of the system.

Acknowledgments

The authors are thankful to Hugo Krawczyk for his many useful comments.

References

[1] R. Blahut. *Theory and Practice of Error Control Codes.* Addison-Wesley, 1984.

[2] C. Boyd. Digital Multisignatures. In H. Baker and F. Piper, editors, *Cryptography and Coding*, pages 241–246. Claredon Press, 1989.

[3] R. Canetti and Amir Herzberg. Maintaining security in the presence of transient faults. In Y. Desmedt, editor, *Advances in Cryptology — Crypto '94*, pages 425–438, Berlin, 1994. Springer-Verlag. Lecture Notes in Computer Science No. 839.

[4] M. Cerecedo, T. Matsumoto, and H. Imai. Efficient and secure multiparty generation of digital signatures based on discrete logarithms. *IEICE Trans. Fundamentals*, E76-A(4):532–545, 1993.

[5] D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology — Crypto '82*, pages 199–203, Berlin, 1982. Springer-Verlag. Lecture Notes in Computer Science No.

[6] P. Chen, J. Garay, A. Herzberg, and H. Krawczyk. A Security Architecture for the Internet Protocol. *IBM Systems Journal* **37**, No. 1 (1998), pp. 42-60.

[7] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. In *Proceeding 26th Annual Symposium on the Foundations of Computer Science*, pages 383–395. IEEE, 1985.

[8] R. A. Croft and S. P. Harris. Public-key cryptography and re-usable shared secrets. In H. Baker and F. Piper, editors, *Cryptography and Coding*, pages 189–201. Claredon Press, 1989.

[9] A. De Santis, Y. Desmedt, Y. Frankel, and M.Yung. How to share a function securely. In *Proc. 26th Annual Symp. on the Theory of Computing*, pages 522–533. ACM, 1994.

[10] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures. In J. Feigenbaum, editor, *Advances in Cryptology — Crypto '91*, pages 457–469, Berlin, 1991. Springer-Verlag. Lecture Notes in Computer Science No. 576.

[11] Y. Desmedt. Society and group oriented cryptography: A new concept. In C. Pomerance, editor, *Advances in Cryptology — Crypto '87*, pages 120–127, Berlin, 1987. Springer-Verlag. Lecture Notes in Computer Science No. 293.

[12] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Advances in Cryptology — Crypto '89*, pages 307–315, Berlin, 1989. Springer-Verlag. Lecture Notes in Computer Science No. 435.

[13] Y. Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–457, July 1994.

[14] 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*, pages 384–393. IEEE, 1997.

[15] Y. Frankel, P. Gemmell, and M. Yung. Witness-based Cryptographic Program Checking and Robust Function Sharing. In *Proc. 28th Annual Symp. on the Theory of Computing*, pages 499–508. ACM, 1996.

[16] Y. Frankel, P. Gemmell, P. Mackenzie, and M. Yung. Proactive RSA. In B. Kaliski, editor, *Advances in Cryptology — Crypto '97*, pages 440–454, Berlin, 1997. Springer-Verlag. Lecture Notes in Computer Science No. 1294.

[17] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Info. Theory*, **IT-31**(4):469–472, 1985.

[18] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. In M. Mavronicolas and P. Tsigas, editors, *11th International Workshop, WDAG '97*, pages 275–289, Berlin, 1997. Springer-Verlag. Lecture Notes in Computer Science No. 1320.

[19] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust and efficient sharing of RSA functions. In N. Koblitz, editor, *Advances in Cryptology — Crypto '96*, pages 157–172, Berlin, 1996. Springer-Verlag. Lecture Notes in Computer Science No. 1109.

[20] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In Ueli Maurer, editor, *Advances in Cryptology — Eurocrypt '96*, pages 354–371, Berlin, 1996. Springer-Verlag. Lecture Notes in Computer Science No. 1070.

[21] R. Gennaro, M. Rabin, and T Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. Manuscript, 1997.

[22] S. Goldwasser and S. Micali. Probabilistic encryption. *JCSS*, 28(2):270–299, April 1984.

[23] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing*, 17(2):281–308, April 1988.

[24] L. Harn. Group oriented (t, n) digital signature scheme. *IEE Proc.-Comput.Digit.Tech*, 141(5):307–313, Sept 1994.

[25] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *1997 ACM Conference on Computers and Communication Security*, 1997.

[26] A. Herzberg, S. Jarecki, H.Krawczyk, and M.Yung. Proactive secret sharing, or: How to cope with perpetual leakage. In D. Coppersmith, editor, *Advances in Cryptology — Crypto '95*, pages 339–352, Berlin, 1995. Springer-Verlag. Lecture Notes in Computer Science No. 963.

[27] K.Hickman. Secure Socket Library. Netscape Communications Corp. `<http://www.mcom.com/info/SSL.html`.

[28] H. Krawczyk. Distributed fingerprints and secure information dispersal. In *Proc.* 13*th ACM Symp. on Principles of Distributed Computating*, pages 207–218. ACM, 1993.

[29] H.Krawczyk. Secret sharing made short. In D. Stinson, editor, *Advances in Cryptology — Crypto '93*, pages 136–146, Berlin, 1993. Springer-Verlag. Lecture Notes in Computer Science No. 773.

[30] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proc.* 10*th ACM Symp. on Principles of Distributed Computati on*, pages 51–59. ACM, 1991.

[31] C. Park and K. Kurosawa. New ElGamal Type Threshold Digital Signature Scheme. *IEICE Trans. Fundamentals*, E79-A(1):86–93, 1996.

[32] M. Rabin. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[33] T. Rabin. A simplified approach to threshold and proactive RSA. To appear in H. Krawczyk, editor, *Advances in Cryptology — Crypto '98.*

[34] R.Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2):120–126, 1978.

[35] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22:612–613, 1979.

A Overview of the Information Dispersal Algorithm

The *Information Dispersal Algorithm* (IDA) uses a linear transformation to convert $m = n-t$ bytes of input into n bytes of output. This transformation is given by an $m \times n$ matrix T over GF(2^8). Moreover, the matrix T has the property that every $(n-t)$ columns of T are linearly independent. Thus, each input and output byte is viewed as an element of GF(2^8). The block size is m bytes and the operation is repeated for every m bytes.

Let the (i,j)th entry of T be represented by $T_{i,j}$. Let $P_0, P_1, ...P_{m-1}$ be a block of input. Then the output bytes $Q_0, Q_1, ...Q_i, ...Q_{n-1}$ are given by

$$Q_i = T_{0,i} \cdot P_0 + T_{1,i} \cdot P_1 + ...T_{m-1,i} \cdot P_{m-1} \ ,$$

where the arithmetic is performed in the field GF(2^8).

Given any m output bytes, the input can be recovered because every m columns of T are linearly independent. In other words, the matrix S formed by taking the columns of T which correspond to these m output bytes is invertible. Again, the inverse of this matrix is computed over GF(2^8).

As an example, let $m = 3$, and $n = 5$. The following matrix T has the property that every 3 columns of T are linearly independent. Note that we are using polynomials in x for representing elements of GF(2^8). The polynomial arithmetic can be done modulo $x^8 + x^6 + x^5 + x^4 + 1$, which is an irreducible polynomial over GF(2).

$$T = \begin{pmatrix} 1 & 0 & 0 & 1 & 1+x \\ 0 & 1 & 0 & 1 & x \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

If for example, only the first, second and fifth byte of a coded text are known, the plaintext (or original text) can be retrieved by applying the following transformation to the three bytes of coded text:

$$\begin{pmatrix} 1 & 0 & 1+x \\ 0 & 1 & x \\ 0 & 0 & 1 \end{pmatrix}$$

(Note that this matrix is its own inverse.) The reader is referred to [32] for further details.

B Example: Threshold RSA

We give a specific example of threshold cryptography assuming that the public key cryptosystem used is RSA [34] and for a specific choice of the public exponent, e.g. 3. In this case, the public encryption key of user U is

$$\mathrm{EK}_U = (3, N) \ ,$$

where N is the RSA module, and

$$\mathrm{DK}_U = (d, N) \ ,$$

where d is the inverse of 3 modulo $\phi(N)$. Assume that the user's secret key DK_U has been shared among the servers as an n-out-of-n sharing, meaning that all the shares will be required in order to reconstruct the key (this is without loss generality, as it is easy to generalize to a threshold scheme). We can assume that server V_j's share of the key is

$$\mathrm{dk}_{U,V_j} = d_j \ ,$$

where

$$d_1 + ... + d_n = d \bmod \phi(N) \ .$$

Assume we want to compute a signature $\sigma = m^d \bmod n$ for a message m. Then each server can compute the following

$$\sigma_j = m^{d_j} \bmod N$$

and then we see that

$$\sigma_1 \cdot \sigma_2 \cdots \sigma_n = m^{d_1} \cdot \cdots m^{d_n} = m^{d_1 + ... + d_n} = m^d = \sigma \ .$$

A dual approach clearly works for RSA signatures.

C Example: Blinding with RSA

Once again we present an example of the blinding technique [5] based on RSA. The server owns the secret key DK $= (d, N)$ and the user knows the public key EK $= (e, N)$. The user wants to decrypt a ciphertext $c = m^e \bmod N$ without telling the server c. The user chooses r at random and computes $s = r^e \bmod N$. The user then gives $cs = (mr)^e \bmod N$ to the server who returns $w = (cs)^d = mr \bmod N$. Finally, the user computes $m = \frac{w}{r} \bmod N$.

D On Alternative Designs

In this section we compare the costs—in terms of computation, storage and communication—of our single connection/gateway design to those of having the user communicate with the servers independently and in parallel. It turns out that besides providing for broad applicability, the single connection design is more efficient both at the user and servers ends.

D.1 Cost at the user

The computation and communication costs for the user in the single connection model is always lower than having the user communicate directly with all (that is, at least $t + 1$ of) the servers, as the tables below indicate. It may seem at first that due to the communication with a single gateway our design requires expensive cryptographic machinery (such as threshold cryptography) to meet the security requirements. However, one should note that the use of threshold cryptography can be removed only at the expense of heavy computations and/or heavy memory requirements at the user's end. Even when communicating with all the servers the user still needs to receive a receipt for the transaction. Thus, he can go one of two ways:

- Have each server sign an individual receipt for the deposit; this in turn requires the user to verify and store n signatures.
- To save on storage, use threshold cryptography (which means no savings on cryptographic machinery); but this requires the user to verify n partial signatures, which typically is a more expensive verification process than verification of a regular signature (for example, in the case of RSA one can use a small verification exponent for regular signatures, but not for partial signatures).

With respect to blinding, we note that the removal of the blinding step would result in the user needing to decrypt n shares of the file key FK, as opposed to a single blinding operation which is an encryption with a small exponent.

Single Connection	**Multiple Connections**	
	Split at user	Split at servers
1 encryption	IDA	secret sharing
2 signatures (expected);	secret sharing	1 encryption
$t+1$ signatures (worst-case)	n signatures	1 signature
1 small signature verification; 1 signature storage	n small signature verifications, if store n signatures OR n big signature verifications, if store 1 signature	

Table 1: Computation and storage costs of Deposit with Confidentiality at the user

Single Connection	**Multiple Connections**	
	Split at user	Split at servers
1 signature verification (small exponent)	n signature verifcations (big exponent)	
1 encryption (small exponent; for blinding)	1 decryption (big exponent)	

Table 2: Computation costs of Retrieval with Confidentiality at the user

For the comparison, we consider the case of Deposit with Confidentiality, where symmetric-key encryption and decryption are also performed. We divide the multiple connections model into two, depending on whether the IDA "split" of the file is done at the user or at the servers. Let us consider first the communication costs. Let B denote the blow-up factor (see Definition 3) and c be a constant. Then the communication cost for Deposit in the case of a single connection is $|F| + c$. On the other hand, it is easy to verify that the cost in the multiple connections model would be $B|F| + nc$ if the IDA split is performed at the user, or $n(|F| + c)$ if performed at the servers.

Table 1 shows the number of operations and storage requirements at the user to perform Deposit with Confidentiality, while Table 2 shows the costs for Retrieval. In short, both tables show that the costs of multiple connections are higher than in our single connection model.

D.2 Cost at the servers

We now analyze the computational/communication requirements on the servers. Again, it may seem at fisrt that if the user communicates with each of the servers directly this should reduce the workload for the servers and the total communication among them. However, if the user communicates with all the servers and they do not perform the "echo" stage, then it is possible for an adversary to create a valid receipt for a file that the servers will not be able to reconstruct later. In turn, this would result in a liability problem for the storage system.

More specifically, consider the scenario of a fraudulent user colluding with t faulty servers. The user sends the file to $t+1$ servers (t faulty and 1 correct), and thus is able to obtain a receipt. In order for the servers to be able to retrieve the file later they will have to either:

- store the whole file (which entails a blow-up factor of n); or
- echo the file so that there is a guarantee that if a correct server has it, then all the other good servers have it as well.

Thus, there are no savings in communication for the servers in the multiple connections model. Notice that in the single connection model this kind of coalitions is not a problem, as this case can be easily reduced to the case where the gateway is faulty.

Under slightly different assumptions about the model, which we will now explain, the attack above is plausible even without the user being malicious. We conceive the communication between the user and the servers to take place on a public network, such as the Internet, while the communication among the servers to be on a more secure "Intranet." If we assume that the adversary can drop or block messages on the public network, but not on the Intranet, then the adversary can have only $t+1$ servers receive it: t faulty and 1 correct; these servers will produce a valid receipt, and then we are back in the situation described above. Once again, this is not a problem in our model since an adversary that blocks messages on the public network can create at most a denial-of-service attack, which is effectively similar to the one of failing ("crashing") the gateway.

Finally, as pointed out in Section 2.8, the communication among the servers (the echo messages) can be further reduced by echoing only a message digest (hash) of the files, and the actual files only if necessary.