Design and Implementation of a Secure Distributed Data Repository

Arun Iyengar Robert Cahn Juan A. Garay Charanjit Jutla
IBM Research, T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598, USA
{aruni,cahn,garay,csjutla}@watson.ibm.com

Abstract

This paper describes the design and implementation of an on-line, distributed data repository, known as an *electronic vault* (e-Vault), which stores information across a network securely so that integrity is preserved even when some of the servers malfunction. After a document has been deposited by a client, the e-Vault returns a receipt to the client which can be used to verify that the document has been properly received by all (correct) servers. The e-Vault disperses information across the servers using Rabin's Information Dispersal Algorithm (IDA). Shared distributed digital signatures are used to provide proof to the client that a document has been correctly received by all servers. Cryptographic hash functions are used to fingerprint information in order to determine if the information is later corrupted. Our e-Vault has been implemented on multiple nodes of an IBM SP2 multiprocessor. Clients communicate with the e-Vault using SSL-enabled browsers. Design extensions for confidentiality of information are also presented.

1 Introduction

There is often a need for storing information securely across multiple machines so that the information can be retrieved in the event that one or more machines fail. In some cases, it may even be desirable to distribute the machines geographically; this would allow information to be retrieved in case a catastrophic event strikes a particular area. Failures can be the result of malicious (e.g., someone breaking into a machine) or nonmalicious (e.g., a machine crashing) events.

When information is distributed across multiple machines, the space taken up by the information can become significant. The most straightforward method to disperse information across multiple sites would be to replicate the information across the sites. This leads to a factor of n blowup in space requirements, where n is the number of sites storing a copy of the information. Several other algorithms for distributing information with less space overhead have been proposed, including the Information Dispersal Algorithm (IDA) by Rabin [12].[1] Another problem with simple replication is that stored information can be obtained if a hacker manages to break into any server containing the replicated information.

IDA allows for the reconstruction of information when a subset of the machines is down. Extensions have been proposed that 1) allow for the reconstruction when information coming from a subset is not just missing, but has been altered, while maintaining the same minimal space overhead [6], and 2) make the storing—not just the reconstruction—of the information possible when a subset of the machines is malfunctioning [4].

This paper describes the design and implementation of an **on-line, distributed data repository**, known as an *electronic vault* (e-Vault), which securely distributes information across several servers using IDA with the extensions mentioned above. A general block diagram of such a system is shown in Figure 1. The system can reliably store and reconstruct information if a minority of servers fail or are compromised. Our system has features for guaranteeing that a client document is properly distributed across several servers, and that the **integrity** of the information is preserved after a document is stored at the servers. In order for hackers to steal the information contained in an e-Vault, they would have to break into a majority of the servers comprising the system. Information is thus preserved more securely than if a single copy is maintained or if copies are replicated across multiple servers for high availability.

The remainder of the paper is organized as follows. In Section 2 we present the general architecture of the e-Vault. In Section 3 we present some performance measures, while in Section 4 we discuss on-going and future work. In Appendix A we present an overview of threshold cryptography.

[1]We present a short overview of IDA in Section 2.5.

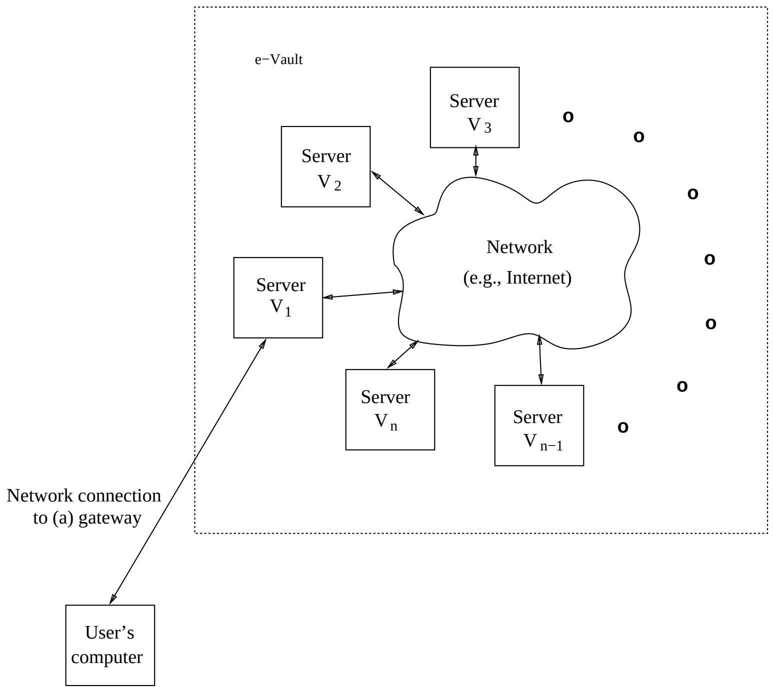

Figure 1: Block diagram of the e-Vault.

2 Electronic Vault Architecture

Figure 2 shows a diagram of our system. While our system contains only three servers, the architecture can be easily generalized to include many more servers (Figure 1). The client communicates with a single server known as the *gateway.* The other server nodes, known as *internal servers,* communicate with the gateway but not with the client. We first provide a description of the basic operation of the e-Vault, and then describe each of the components in detail.

2.1 Operation of the e-Vault

When a request to store information (a *deposit*) is received by the gateway, the gateway distributes the information to all internal servers. Each server applies IDA to the file and stores an IDA *slice* (plus some additional information—described later) within its memory. In order to retrieve the document, the gateway contacts enough servers to obtain a sufficient number of IDA slices to reconstruct the file.

In order to guarantee that a sufficient number of servers have received correct information when a client makes a deposit, distributed digital signatures [3] are used (see Appendix A), meaning that each server contributes a *partial* signature on the (hash of the) information received from the client. The gateway combines the partial signatures from servers receiving information and sends the combined signature to the client for verification. In the current system, all servers must send a correct partial signature in order to generate a valid combined signature (a *receipt*). In the future, a majority of correct partial signatures will be sufficient to generate a valid receipt (see Section 4). That way, the number of correct partial signatures needed by the gateway to generate a correct distributed digital signature can be set as a parameter.

When a client requests the e-Vault to retrieve information, some of the IDA slices may have been corrupted. In order to detect this and throw out bad IDA slices, each server performing IDA during a deposit computes and stores fingerprints (hash digests) for all IDA slices. During a retrieval, the servers send to the gateway not just their IDA slices but also the hash digests; the gateway determines which IDA slices are valid by applying majority to the stored hash digests. Clients also have the option of computing a hash digest for deposited information at the time of a deposit. Upon retrieval, the client can compare the hash of the retrieved document to the hash of the deposited document to make sure that they are the same.

2.2 e-Vault **implementation details**

Clients communicate with the e-Vault via an SSL-enabled browser. The gateway invokes server programs in response to client requests via the Common Gateway Interface (CGI) [15]. Information passed between the client and server is encrypted via SSL. An SSL-enabled browser is sufficient for a client who is satisfied with a base level of functionality and is not interested in sophisticated security features. More advanced security is provided by programs which execute on the client and perform features such as verification of distributed signatures returned by the gateway and verification of retrieved documents via hash functions.

The gateway communicates with internal servers via a proprietary protocol. Long-running processes known as *evault daemons* run on each internal server and listen for communications from

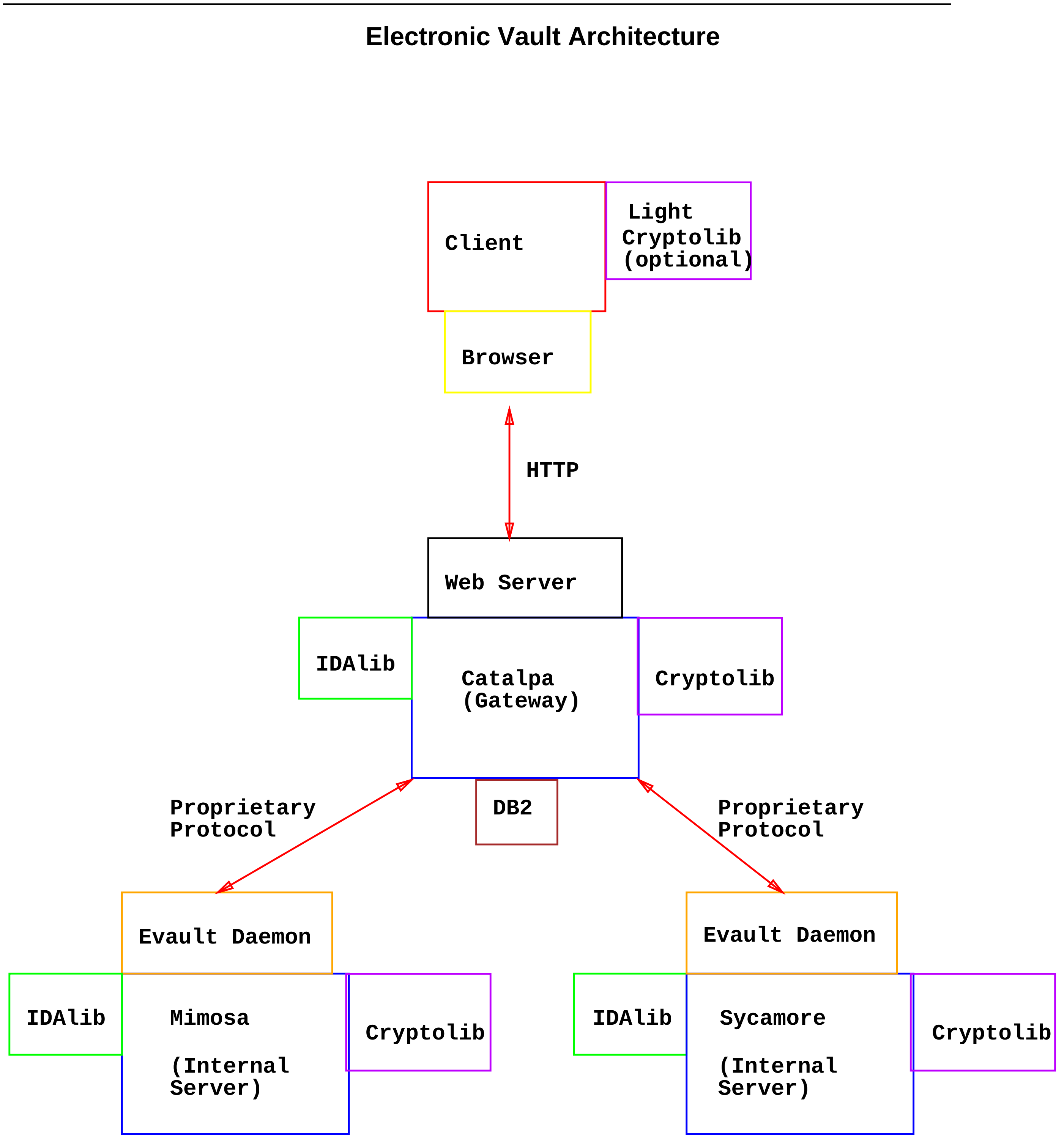

Figure 2: The electronic vault architecture. Although our implementation only contains three servers, the architecture can be generalized to contain significantly more servers.

the gateway. While it may be possible to implement much of the functionality of evault daemons using Web servers, evault daemons have significantly less overhead than Web servers. The gateway can make a single open connection with an evault daemon and make several transactions with an internal server over the single connection. By contrast, Web servers often result in multiple connections for multiple transactions which seriously hurts performance. Evault manager daemons are multithreaded and do not spawn off new processes in order to invoke internal server functions. By contrast, most Web server interfaces for invoking server programs result in higher overheads (e.g. CGI). The proprietary protocol which the gateway uses to communicate with internal servers requires less overhead to process than the `http` protocol.

All of the e-Vault servers have cryptographic libraries which perform functions such as public key operations, computing partial/distributed signatures, and computing hash functions. Each node also has an IDA library which both splits up and recombines documents using IDA.

The gateway stores information about clients (e.g., names, addresses, phone numbers) and transactions using IBM's DB2 relational database. Communication between servers comprising the e-Vault may optionally be secured (e.g., using IPSec protocols) if the servers are communicating over a public or insecure network.

The e-Vault possesses a public and private key pair. The private key is split and distributed across all the servers in a way suitable for performing the distributed/threshold cryptographic operations.

When clients register with the electronic vault, they provide information such as names, contact information, user ID's and passwords. Information passed between clients and the e-Vault is encrypted via SSL. The e-Vault stores client information within the database.

When a client deposits a document with the e-Vault, the document is encrypted via SSL as it passes from the client to the gateway. The gateway sends the document to each internal server. Each server applies IDA to split up the file into slices. The slices are fingerprinted using a one-way hash function. Our electronic vault generates 32-byte fingerprints using MD5; fingerprint sizes are independent of the size of the data being fingerprinted. Each server stores one IDA slice and maintains fingerprints for all IDA slices. As mentioned before, the fingerprints are subsequently used to determine which of the IDA slices are valid during a retrieval. The client may optionally store a hash of the deposited document for verification purposes at retrieval.

When a client retrieves a document from the e-Vault, the gateway receives the request and obtains enough good IDA slices from the various servers in order to reconstruct the document. The gateway determines which IDA slices are good from the fingerprints which were stored for each slice at the time the document was deposited. The gateway reconstructs the document using IDA and sends the document to the client. The client may optionally verify that the returned document is correct by hashing it and comparing the hashed value to the one stored when the document was deposited.

2.3 Client architecture

There are two client architectures: minimal and sophisticated. The minimal client is appropriate when support is limited and the degree of automation needed is slight. Typical users of the minimal client are remote users who need to back up a few important files against loss. Such users can be expected to know which files need to be stored on the e-Vault and are willing to send them to the e-Vault by a manual process.

2.3.1 The minimal client

The minimal client consists of three programs. The first, `extractkey`, parses the registration form and extracts the e-Vault's public key and modulus. This is needed if the user is going to verify deposits. The next program, `verify`, has the job of scanning the reply from a deposit and verifying the signature using the public key and modulus stored by `extractkey`. Also, it optionally stores the name and hash of a file so that the restored file can be verified when retrieved from the e-Vault. The last program, `extractfile`, has the job of extracting the file from the HTML page when it is retrieved and verifying the hash against the value stored by `verify` when it was deposited. For this purpose, we use an MD5 hash of a file since it has the standard collision-resistant properties which are desired for detecting alteration or tampering.

Documents to be stored at the electronic e-Vault initially exist in a file at the client. HTML 2.0 allows the user to define a form with an element of type `file`. This form element allows the user to type in the name of the file to be deposited. The browser uses this form to send the entire contents of the file to the Web server which reads the contents of the file via standard input.

2.3.2 The sophisticated client

The sophisticated client does not rely on the browser to upload and download files. Rather than being HTML page-based, it uses standard file chooser GUI elements and hides the HTML pages from the user. The simplest version of operation allows the user to select a set of files from a file chooser. When the list is complete, the sophisticated client computes hash values for each file. The sophisticated client contacts the gateway and sends each file to the e-Vault. The e-Vault returns signatures for each file which are all verified by the client to insure that all files were successfully deposited. Similarly, when files need to be retrieved, the user can specify multiple files using a file chooser-like interface. The client then contacts the e-Vault, retrieves all files, and verifies that all retrieved files are correct from the hash values stored when the files were deposited.

In addition to manual selection, the sophisticated client offers automation; using profiles and archive bits, like standard backup processes, it can create a list of files. For example, it is possible to specify that all the files matching the template *.c and *.h in the directory `/important/new/application` should be backed up to the e-Vault but the object files should be ignored. In addition, version control allows the client to specify in a profile the number of copies of a file to be maintained at the server. For example, we might set up the profile to maintain the 5 latest versions of any c source file, an unlimited number of versions of the file `my_last_will_and_testament` but only the last version of the file `todays_shopping_list` in the archive.

As mentioned in Section 4, another function of the e-Vault is **confidentiality**. Thus, another function of the sophisticated client is file encryption. The e-Vault architecture allows the user to encrypt a file with a randomly generated symmetric key and then deposit the encrypted file and the key on the e-Vault in such a way that the server cannot use the key to decrypt the file and reveal the contents. The client code handles the issues of key generation, encryption, and file translation into printable ASCII. Some encrypted files are more conveniently maintained in printable ASCII. Consequently, the file can be nibblized and encoded 2-for-1. A slightly better solution is for 3 bytes to be broken into 4 6-bit fragments and each be transmitted as a printable character producing a 4-for-3 encoding.

2.4 Cryptographic Library

The novel cryptographic tool used in the e-Vault is the shared distributed digital signature which is used to generate *receipts* for clients' deposits. We give here the specific RSA-based shared signature that is implemented in this system; see Appendix A for additional background information. Let the public key of the e-Vault be

$$\mathrm{PK} = (3, N) ,$$

where N is the RSA module and 3 is the public exponent, and the secret key be

$$\mathrm{SK} = (d, N) ,$$

where d is the inverse of 3 modulo $\phi(N)$. The secret key SK is shared in an n-out-of-n scheme (i.e., all the shares will be required in order to reconstruct the key). Let the secret key share (partial key) of server j be

$$\mathrm{SK}_j = d_j ,$$

such that

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

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

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

and forward it to the gateway. The gateway computes the distributed signature by multiplying the partial signatures $\bmod N$.

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

In our system we use AT&T's `Cryptolib` [7] to implement the basic cryptographic functions. We have also extended the library to implement partial key distribution, partial signature generation, and combining the partial signatures. One of the optimizations in `Cryptolib` to compute the RSA signature is to use the Chinese remainder theorem data (essentially $\phi(N)$). However, partial secret keys cannot contain this data, for that would give the holder of the partial key the capability to easily compute the full secret key. Thus, this optimization cannot be employed to compute the RSA partial signatures.

2.5 IDA Library

First, we present a short overview of the Information Dispersal Algorithm (IDA) and then the design for the case of five servers, and comments on our implementation for three servers. Let n be the total number of servers and t the maximum number of servers that may fail (crash). 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; also note that computing $y \cdot (1+x)$ in GF(2^8) is rather easy. However, other entries in the inverse of 3×3 submatrices of T are more complex, and each such multiplication should be hand-coded for maximum efficiency.

For the case of three servers with at most one faulty server, the field GF(2) suffices (i.e., XOR is the only operation required). The IDA library for our implementation is thus a simple collection of functions to perform the XOR of two bytes of the file at a time, saving the corresponding bytes according to the server's ID, and functions for the respective reconstruction.

3 Performance

We have measured the performance of the electronic vault using the WebStone benchmark [14]. WebStone is a widely used benchmark from Silicon Graphics, Inc. which measures the number of requests per second which a Web server can handle by simulating one or more clients and seeing how many requests per second can be satisfied during the duration of the test. Each node of the electronic vault is an IBM RS/6000 Model 590 workstation containing a 66 Mhz POWER2 processor and running AIX version 4.2.0.0. Clients communicated with the e-Vault over an Ethernet.

The graph in figure 3 shows the number of retrievals per minute which the electronic vault can sustain as a function of document size. Performance was limited by the electronic vault servers and not by network bandwidth. For documents containing less than 10 Kbytes, the performance bottleneck is the database contained at the gateway. Each program invoked by the gateway which accesses IBM's DB2 database in order to satisfy a client request must make a new connection to DB2. Connecting to DB2 is expensive and constitutes the principal bottleneck. The database overhead does not vary much with document size, however. As document sizes increase beyond 10 Kbytes, performance overhead becomes dominated by copying, cryptographic operations, and communication. The retrieval rate exceeds one document per second until document sizes exceed 50 Kbytes.

An optimization which would improve performance considerably for small documents would be to maintain long-running processes with persistent open connections to the database [8]. At this point, it might be possible to further improve performance by using faster interfaces than CGI such as NSAPI [10], ISAPI [9], ICAPI by IBM, or FastCGI [11]. Performing the latter optimiza-

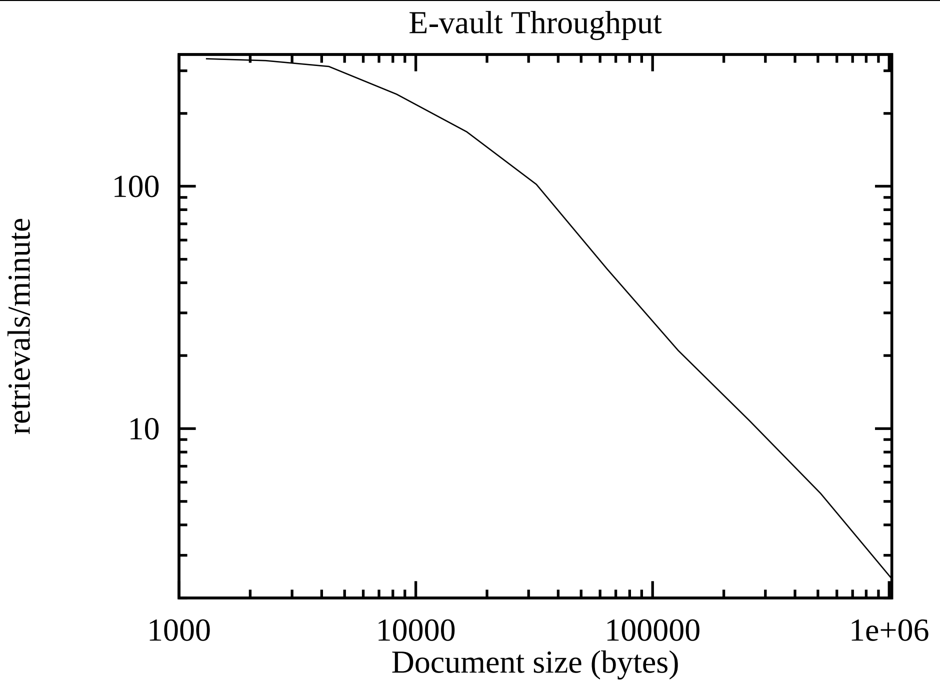

Figure 3: The number of retrievals per minute which the electronic vault can sustain as a function of document size.

tion without the former would result in little performance improvement because the overhead for connecting to DB2 is considerably higher than the overhead for invoking programs via CGI.

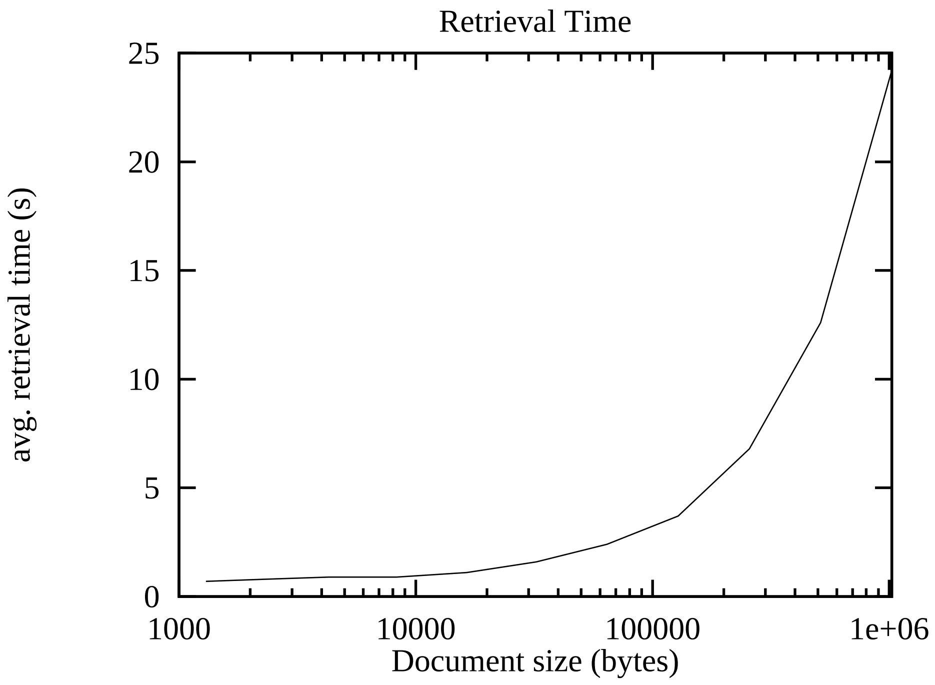

Figure 4: The average time for a client to retrieve a document as a function of document size.

Figure 4 shows the average retrieval time seen by the client as a function of document size for a lightly loaded system. Network bandwidth between the client and server was never a bottleneck.

4 Extending the Electronic Vault Architecture

In this section we present several directions in which the e-Vault architecture can be extended; this is the subject of current work.

4.1 Uniformity

In the e-Vault architecture presented in Section 2, the client interacts with a *single* server, the gateway. This design decision was based on efficiency and simplicity reasons: comparable security properties can be achieved if the client interacted with all the servers—not just the gateway, but at the cost of establishing several `http` connections simultaneously; we wanted to limit the added functionality that current browsers would need in order to be able to use the e-Vault. However, in our implementation, the gateway is always the same machine (Catalpa). In order to fully satisfy the claimed integrity and fault tolerance properties, in the final design the clients will interact with a single, but not necessarily the same server. This implies the replication (and consistency maintenance) of the DB2 modules across the servers, as well as time-out mechanisms that would direct the client to contact a different gateway when one is not responding.

4.2 Confidentiality

An added functionality of the e-Vault is the **confidentiality** of information, by which we mean that any collusion of a minority of the servers (except ones including the client) should not be able to learn anything about the information that is deposited. Confidentiality is easily achieved by encryption. Yet, this in return 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 deposited. Under this scheme, how would users be able to retrieve their files confidentially? Remember that they are communicating with the system through a single gateway, which means that if only the known techniques of secret sharing reconstruction were used [13, 2], then the gateway would manage to know all the information available to the users. In the full design, we implement the ideas of [4] which combine threshold cryptography and *blinding* techniques [1] in a novel way to achieve the confidentiality requirement stated above.

In a nutshell, the client generates a symmetric (e.g. DES) key and stores the encrypted file (with symmetric key) and encrypted key (with client's public key) using the deposit protocol described in Section 2. At retrieval time, the user generates a *blinding* factor, encrypts it with its public key and sends it to the e-Vault. Each server, using its share of the client's secret key, performs a (partial) decryption of the *product* of the encrypted blinding factor and encrypted file key and sends it to the gateway. The gateway then decrypts the product and sends it to the client. The client factors out the blinding factor to recover the encrypting key for the file.

Another benefit of this approach—besides preventing the gateway from learning anything about the contents of the information—is that at no time is sensitive cryptographic key material stored at the client; only the blinding factor is kept for a short period of time [4]. We already discussed in Section 2.3.2 the modifications to the client to perform file encryption; the modifications to the servers to perform the above operations are minimal.

4.3 t-out-of-n signature scheme

As mentioned in Section 2.4, the distributed signature scheme we have implemented is n-out-of n, meaning that all servers must be up in order to generate a valid receipt. At this time, our lab, in conjunction with another organization, is implementing a threshold DSS signature scheme [5]. In other words, only $t + 1$ servers will be needed to generate a valid signature (recall that in our design, $n > 2t$). The incorporation of this module will require modifications to the Cryptographic Library, as well as to the evault daemons, as several additional exchanges are necessary among the servers in order to satisfy some of the scheme's zero-knowledge properties.

Acknowledgments

Tal Rabin and Rosario Gennaro contributed to the ideas behind the e-Vault. The e-Vault daemon uses code written by Jim Challenger and Russ Miller. Louise Iyengar designed the e-Vault home page.

References

[1] 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.

[2] 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.

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

[4] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retreival. 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.

[5] 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.

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

[7] J. Lacy, D. Mitchell, and M. Blaze. Cryptolib 1.2. http://www.homeport.org/adam/crypto/cryptolib.phtml.

[8] Y. H. Liu, P. Dantzig, C. E. Wu, and L. M. Ni. A Distributed Connection Manager Interface for Web Services on SP Systems. In *Proceedings of the International Conference for Parallel and Distributed Systems*, June 1996.

[9] Microsoft Corporation. (ISAPI) Overview. http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimrg.htm.

[10] Netscape Communications Corporation. The Server-Application Function and Netscape Server API. http://www.netscape.com/newsref/std/server_api.html.

[11] Open Market. FastCGI. http://www.fastcgi.com/.

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

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

[14] Silicon Graphics, Inc. World Wide Web Server Benchmarking. http://www.sgi.com/Products/WebFORCE/WebStone/.

[15] Various. Information on CGI. hoohoo.ncsa.uiuc.edu:80/cgi/overview.html, www.yahoo.com/Computers/World_Wide_Web/CGI___Common_Gateway_Interface/, www.stars.com/, and www.w3.org/pub/WWW/CGI.

A 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 [13]. Let K_i be the share given to player P_i. [2] 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 SK which is the inverse of a public key PK.

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 PK (i.e., $c = \mathbf{E}_{\mathrm{PK}}(m)$ 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 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

[2] 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.

A survey of threshold cryptography techniques can be found in [3].