The Strong Eternity Service

Tonda Beneš

Faculty of Mathematics and Physics, Charles University Prague

Abstract. Strong Eternity Service is a safe and very reliable storage for data of high importance. We show how to establish persistent pseudonyms in a totally anonymous environment and how to create a unique fully distributed name-space allowing both computer-efficient and human-acceptable access. We also present a way how to retrieve information from such data storage. We adapt the notion of the mix-network so that it can provide symmetric anonymity to both the client and the server. Finally we propose a system of after-the-act payments that can support operation of the Service without compromising anonymity.

1 Introduction

We completely re-think the structure of the system called 'Eternity Service' [1] introduced in 1996 by Ross Anderson. We introduce cryptography and other techniques to strengthen the resistance of the system. To distinguish our proposal from the original one and from other clones [2, 3] we name it 'Strong Eternity Service'[1]. We summarise our most interesting ideas about construction of the Service here.

2 Threat Model

We allow a would-be attacker to employ any means and methods of attack even if they violate ethics (bribery, extortion), human rights (physical violence) or require large resources (human resources, money, technology, skills, time). The attacker can abuse various institutions to create political, legal, social or religious pressure against the owner of the data or system maintainers. We assume that a determined and skilled attacker can gain access to data of his interest virtually to the same extent as the authorised user.

We classify possible threats into four categories:

Blunt influences do not understand the content of managed data nor they can interpret internal states of the system. Their behaviour is stochastic. They can occasionally influence large parts of the system at once. Vandals, thieves, technical faults, natural disasters, wars, epidemics etc. belong to this category.

Amateur opponent understands the structure and the current state of the assaulted system. He can interpret stored information. His actions are primarily

[1] In further text we will use the abbreviation 'Service' to refer to our system.

targeted against stored data or the system itself or its components. Especially dangerous can be long-term simultaneous influence of many such attackers on different parts of the system. Various types of hackers, crackers, disloyal employees of system maintainers, small interest groups etc. belong in this category.

Professional opponents have roughly the same characteristic as the amateur ones. Their most dangerous feature is the ability to concentrate large resources to a single task. Large companies, intelligence services, armies and similar organisations are the most usual threats belonging here.

The goals of *authorities* are roughly the same as in the case of both amateur and professional opponents. Unlike them, authorities must not act latently. Using generally obligatory orders they can influence large parts of the system without even knowing who operates which server or where the server is physically located. This category consists of courts, governments, political or religious leaders etc.

3 Goals and Means

The goal of our work was to propose a system with the following features:

1. It is very important to ensure highest possible availability.
2. The system should offer a high degree of reliability
3. It has to provide high information survivability, especially in the event of huge damage.
4. The system should protect stored information at least from all the threats listed in Sect. 2.
5. The degree of achieved protection should depend on client requirements.

In the rest of this section we discuss various features of the proposed system that ensure the above goals and means how these qualities are achieved.

Unlimited Availability The requested high data availability means that the system itself has to be sufficiently available. However we can not prevent any opponent from attempting to prevent clients from accessing their data. All we can do here is to make these attempts ineffective and pricy.

Our system should have as many mutually equivalent entry points as possible. Communication between nodes should prevent selective attacks. Another protection arises from the use of a widely employed communication platform (that can not be blocked completely) and from the fact that system uses no 'magic numbers' such as well-known ports, addresses etc.

High survivability The stored data should remain available despite the extent of the damage that the system suffers in the long run. System should be resistant both to long-term influence by active opponents and to temporary or permanent loss of its large parts.

Extendibility The system size is one of the best defensive mechanisms available. Provisions allowing fast and simple spread of the Service are very important. At the same time, all parts of the system should be mutually independent. In particular we have to avoid any registration or certification process.

Fully distributed design Any centralised part represents a very attractive target for a focused attack. Our system should avoid any centralised parts at almost any price. If the system uses services from external providers it is necessary that sufficiently many alternative providers are available.

Forward secrecy Strict control of processing residues is crucial. If no history is available, an attacker has to analyse the system on-the-fly. Any component that falls under the opponent's control does not bring any useful information to him.

Good habits It is important that Service operation does not cause any inconvenience to the surrounding environment. Individual parts of our system should co-exist smoothly with elements of network infrastructure.

Other features Other features relate to techniques used to implement the Service components rather that to the behaviour of the final system.

4 Service Structure and Features

The Service is a fully distributed system consisting of a (preferably) great number of servers of several types. These servers are spread around the world and their physical location is kept secret. The Service does not define any identity. The system is fully anonymous, there is no notion such as 'owner' or 'authorised user'. Anybody is authorised to perform all available operations, nobody's identity is queried. This is why we use the term 'client' rather than 'user'.

The client stores his information in several redundant copies. The number of copies determines the achieved level of security. The client should select the servers actually holding the copies at random. He should not keep any record of which servers he contacted.

The system is vertically divided into two layers. The bottom layer is a mission independent anonymous routing mechanism. The upper layer is made up of servers that perform services for the clients and carry the whole functionality of the Service. This design allows us to share the lower layer with other systems.

The key features of the Service are achieved through a careful design of individual servers. The description of individual servers is far beyond the scope of this text. We outline global behaviour of the system here.

4.1 Server Types

We use these types of servers:

- *Mix Server (MX)*—realises all functions connected with message transport, i.e. receives messages to be transported, divides them into transferred datagrams, transports these datagrams and re-collects the original messages at the recipient's side. Mix servers provide support for addressing.

– *Eternity Server (ES)*—carries out the functions of the Service provided to clients. ES receives and stores a client's data, and searches and retrieves the data in accordance with the client's requirements.
– *Bank Server (BS)*—supports operations connected with the system of payments from clients to servers for the provided services. These servers provide an important interface to banking institutions supporting the Service.
– *Certificate Server (CS)*—lower-level transport mechanism makes use of several types of certificates used to properly address the counterparts of a transaction and to construct their addresses. CS concentrates such certificates and makes them publicly available.
– *Client Module (CM)*—is not a regular server. Rather than that, it serves to the client as an interface allowing him to properly contact the Service, issue requests and receive responses.
– *Eternity Proxy Server (EPX)*—is an optional part of the Service that further makes it easier to contact the Service. It can provide an easy-to-use web based interface, and it can allow to clients with restricted access to the Internet (dial-up, e-mail-only etc.) to use the Service.

Note that all servers of one type are functionally fully equivalent, i.e. it is insignificant which particular server the client contacts. There is no hierarchy between servers of one type, the system does not use any notion of 'neighbourhood' or 'distance'. Any server can communicate with any server of his choice, all co-operating parties should be selected at random.

If anybody wishes to join the Service, he simply sets-up a new server, issues the appropriate certificates to allow others to contact him and starts operating without any notification. Similarly, revocation of certificates makes the server unreachable and it can disappear quietly.

4.2 Provided Functions

A proper selection of performed operations is one of the most important protective measures the Service uses. In the view of our considerations in Sect. 2 we excluded all operations allowing modification or deletion of the data:

Store Request Client contacts an ES selected at random and requests it to store data for specified period. If server complies, client supplies the data and a keyword-list associated with the data, which characterises it, and pre-pays storage fees. Server stores the data, upon request passes it to anybody requesting it and after the agreed period removes the data automatically. The storage fees are transferred to the server after this period.

Find Request Client supplies a description of the requested data. The contacted server first of all searches its own data structures and subsequently forwards the request to several colleagues selected at random. Depth of search is controlled as well as the total size. The server subsequently summarises all obtained responses and passes them to the requestor. Each record about matching data contains a unique identification of the data—ICK (internal checksum).

Data Request Client identifies the requested data by its corresponding ICK. Servers locate the data by a recursive search in the same manner as in the case of Find Request processing. When the first copy of the requested data is located, the server sends it to requestor and stops any further processing of the request.

4.3 Message Transfer

The Service message transport mechanism (ERM) is based on the idea of a Mix network [6, 4, 9, 10, 5]. We do not introduce any metrics and thus all nodes are equally distant. Our ERM provides mutual anonymity to both sender and recipient. Anybody wishing to communicate using ERM has to issue a special data structure called *Access Certificate (AC)*. The AC describes the path of message transfer across several nodes—Mixes—and finally to the issuer. The sender first gains recipient's AC, and adds to the path description contained in the AC several additional layers.[2]

Messages then consist of three parts. The first one is an *path description* that describes the path of the message transfer from the sender to the recipient, the second one is a *route pack* that prevents some attacks against message transfer and the last one it the *data part*, that contains the useful transferred data.

Access Certificate An *Access Certificate* of server $\mathcal{A}$ is of the form[3]:

$$\langle\!\langle \mathrm{A}_{n/2+1} \big| \mathrm{A}_{n/2} \big| [\mathsf{S}_{n/2} \big| \| \mathsf{Kpbl}_{\mathcal{M}_{n/2+1}} \| \big| \mathrm{A}_{n/2-1} \big| \ldots \big| \mathrm{A}_1 \big| [\mathsf{S}_1 \big| \| \mathsf{Kpbl}_{\mathcal{M}_2} \| \big| \wr$$
$$\wr \mathrm{A}_0 \big| [\textsc{last} \big| \| \mathsf{Kpbl}_{\mathcal{M}_1} \| \big| \mathit{Srv\_Id}; \mathit{CrtId}]_{\mathsf{Kpbl}_{\mathcal{M}_0}}]_{\mathsf{Kpbl}_{\mathcal{M}_1}} \cdots]_{\mathsf{Kpbl}_{\mathcal{M}_{n/2}}} ; \wr$$
$$\wr \mathsf{Kpbl}_{\mathcal{A}}; \mathit{SrvInfo}; \mathit{RevInfo}; \mathit{SrvData} \rangle\!\rangle_{\mathsf{Kprvs}_{\mathcal{A}}}$$

Here Kpbl_i is a public key of Mix i, A_i is its address and S_i is the symmetric key which the Mix i will use to encrypt the client data part before sending the message to the next node. $\mathsf{Kpbl}_{\mathcal{A}}$ is a public key of the target application-level server $\mathcal{A}$ used for encryption. *Srv_Id* is an identification of the target server and the *Crt_Id* is server-wide identification of the certificate because the server can issue several certificates simultaneously. The fields *SrvInfo*, *RevInfo* and SrvData contains basic information about target server, a revocation mechanism data and an additional application-dependant data about the target server. ERM does not interpret these structures, they are used by application-level servers.

[2] The sender appends his own AC to the message to allow the recipient to respond.

[3] Kpbl and Kprv constitute a pair of corresponding public and private keys, S is a symmetric key. $\big|$ denotes concatenation , ; simply separates two independent parts of a message. $\|m\|$ denotes application of a message digest function to the message m. $[m]_{\mathsf{kpbl}}$ denotes sealing with a public key kpbl, i.e. operation $\{\mathsf{k}\}_{\mathsf{kpbl}} \big| \{m\}_{\mathsf{k}}$ where k is a randomly generated symmetric session key. $\{m\}_{\mathsf{k}}$ denotes encryption of message m with key k. We use the $\wr$ sign to indicate line break. $\langle\!\langle m \rangle\!\rangle_{\mathsf{Kprvs}}$ denotes message m with appended signature with key Kprvs.

The whole certificate is digitally signed and the resulting sign *Sign* is appended to its end. For a description of possible signature creation without compromising the security of the protocol see 4.7.

Operation of ERM Each Mix performs uniformly despite its position along the path these steps:

1. The Mix strips its own address from the path description.
2. Subsequently it decrypts the path description part of the datagram. Together with the address of the next Mix it obtains a symmetric key and a digest of the previous Mix public key.
3. The Mix checks that the obtained digest corresponds to the public key received within the route pack.
4. Using the public key from the route pack Mix verifies the integrity of the first part of the route pack.
5. The server checks whether this particular datagram (identified by *Chain* record in the route pack was not recently transferred.
6. If none of the tests performed in steps 3, 4, and 5 fails, Mix continues with the following steps. Otherwise it discards the datagram immediately.
7. If the server detects the inner-most layer of path description it performs steps necessary to complete the corresponding application-level message and to pass the message to an appropriate application-level server. Otherwise server continues with the following steps.
8. The Mix uses the symmetric key obtained in the step 7 to encrypt the client data part of the datagram.
9. It also pads the first part of the datagram to the original length.
10. The Mix replaces the original public key in the route pack with its own and re-signs the first part of the route pack with the corresponding private key.
11. The Mix sends the datagram to the next node.
12. The Mix destroys all the data associated with the transaction.

The recipient knows all the symmetric keys used to encrypt the message during the second part of the path and thus can remove the encryption. Sender has to decrypt (in reverse order) the message before sending it with all the symmetric keys used within the first part of the path.

The Mix scrambles the order of incoming and outgoing messages. Additional protection is provided by the creation of padding traffic (see later).

Here is a message just prepared by the sender[4]:

$$\overbrace{\mathrm{A}_n\big|\lceil\mathsf{S}_n\big|\|\mathsf{Kpbl}_{\mathcal{M}_{n+1}}\|\big|\mathrm{A}_{n-1}\big|\lceil\mathsf{S}_{n-1}\big|\|\mathsf{Kpbl}_{\mathcal{M}_n}\|\big|\ldots\big|\mathrm{A}_0\big|\lceil\diamond\big|\|\mathsf{Kpbl}_{\mathcal{M}_1}\|\big|\mathit{SrvId};\mathit{CrtId}\wr}^{\textit{path description}}$$
$$\wr\rceil_{\mathsf{Kpbl}_{\mathcal{M}_0}}\cdots\rceil_{\mathsf{Kpbl}_{\mathcal{M}_{n-1}}}\rceil_{\mathsf{Kpbl}_{\mathcal{M}_n}}\big|\wr$$

[4] $\{m\}_{\mathsf{k}}^{-1}$ denotes decryption of message m with key k.

$$\underbrace{\wr\langle\!\langle \mathit{CutInfo};\mathit{Chain};\mathit{Time}\rangle\!\rangle_{\mathsf{Kprv}_{\mathcal{M}_{n+1}}}\big|\mathsf{Kpbl}_{\mathcal{M}_{n+1}}}_{\textit{route pack}}\underbrace{\big|\{\{\ldots\{\{\mathit{data}\}_{\mathsf{Kpbl}_{\mathcal{B}}}\}^{-1}_{\mathsf{S}_{n/2+1}}\ldots\}^{-1}_{\mathsf{S}_{n-1}}\}^{-1}_{\mathsf{S}_n}}_{\textit{client data}}$$

The same message after processing at first Mix looks as follows:

$$\mathrm{A}_{n-1}\big|[\mathsf{S}_{n-1}\big|\|\mathsf{Kpbl}_{\mathcal{M}_n}\|\big|\ldots\big|\mathrm{A}_0\big|[\diamond\big|\|\mathsf{Kpbl}_{\mathcal{M}_1}\|\big|\mathit{SrvId};\mathit{CrtId}]_{\mathsf{Kpbl}_{\mathcal{M}_0}\ldots]}\mathsf{Kpbl}_{\mathcal{M}_{n-1}}\big|\wr$$
$$\wr\mathit{padd}\big|\langle\!\langle \mathit{CutInfo};\mathit{Chain};\mathit{Time}\rangle\!\rangle_{\mathsf{Kprv}_{\mathcal{M}_n}}\big|\mathsf{Kpbl}_{\mathcal{M}_n}\big|\{\ldots\{\{\mathit{data}\}_{\mathsf{Kpbl}_{\mathcal{B}}}\}^{-1}_{\mathsf{S}_{n/2+1}}\ldots\}^{-1}_{\mathsf{S}_{n-1}}$$

Each block A_i contains following information:

- protocol number,
- used encryption algorithm and relevant parameters,
- identification of the used public key (a Mix could issue several MC-s).

Thus each address block has an internal structure that will look as follows:

$$\mathit{Addr}\big|\mathit{Prot\_Num}\big|\mathit{Alg\_Num}\big|\mathit{Params}\big|\mathit{Key\_Id}\big|\mathit{Misc}.$$

To make the ERM more robust, we use a multiple Mix at each point of the path. There is one level of path description part of the message:[5].

$$\bigvee_{j=1}^{k}(\mathrm{A}_{i+1}{}^{(j)})\big|[\mathsf{S}_{i+1}\big|\ldots]_{(\mathsf{Kpbl}_{\mathcal{M}_{i+1}{}^{(1)}},\ldots\mathsf{Kpbl}_{\mathcal{M}_{i+1}{}^{(k)}})}$$

All necessary information about Mixes comes from their *Mix Certificates*. The Mix Certificates contain any information about Mixes used to construct the path description. The necessary certificates are obtained from Certificate Server.

Padding Traffic When only a few real messages are available so that real ones can not be mixed with others the server creates an appropriate number of padding messages to cover real traffic among them.

All outbound messages are placed to a structure called the *Send Pool*. A special loop called *Sender* sends the prepared messages.

An instance of the padding algorithm described bellow has to be performed with each inbound message. The variable *srv.Stratum* has to be created with each instance of the algorithm. Also, each padding message contains the field *msg.Stratum*. The `Max_Stratum` and `Padd_Num` are server-wide parameters.

Let an *Empty Send Pool position* be a position within the Send pool currently containing no message. A *Free Send Pool position* is either the Empty Send Pool position or a position containing a padding message prepared to be sent.

[5] $\bigvee_{i=1}^{k}(m_i)$ means k consecutive repetitions of message m_i where each message has the same internal structure, i.e. $m_1;m_1;\ldots;m_k$.

1. If the incoming message *Msg* is a real one, set $srv.Stratum = \texttt{Max\_Stratum}$; otherwise set $srv.Stratum = \min(msg.Stratum - 1, \texttt{Max\_Stratum})$.
2. If the *Msg* is a real message to be forwarded to another node, then place it into a Free Send Pool position selected at random.
3. Create `Pad_Num` of new padding messages so that $msg.Stratum = srv.Stratum$ for each message. The messages are placed at randomly selected Empty Send Pool positions. Extra messages are discarded.
4. If the *Msg* is a padding message or a real message that will not be forwarded elsewhere, create yet another padding message and place it at a random Empty Send Pool position if available.
5. If the *Msg* is a padding message then discard it.

The Sender loop goes round through the Send Pool in a round-robin manner, sends the message if any, and marks the corresponding position as empty.

4.4 Data Deposition and Payments for Provided Services

The client's data is stored so that nobody, including the operators of the servers and clients, can locate it. Missing support of deletion and modification further strengthens protection against direct attacks against any specific piece of data.

The location of each Eternity Server is protected by ERM. The client selects the server where he deposits his data at random. The required level of protection determines how many redundant copies the client stores.

The actual deposition is done in four steps:

1. The client selects a server and issues a request where he specifies the size of data, the requested deposition time and a set of *Authorisations* (see later).
2. The server decides whether it wishes to accept and store the data. If so, it allocates the necessary space and responds to the client. It also indicates the price of storage and the bank that will handle the payment.
3. If client complies with the price he transfers money to the bank's *Pursue Account*. Within some limited time the client has to submit the data to the server along with a proof that he has actually paid the requested amount.
4. The server verifies the proof and if everything is O.K., it stores the data.

From this moment the data is available to anybody requesting it. After the agreed period of data storage has expired the server deletes it automatically.

The server receives fees for provided storage from Pursue Account after it proves that it still holds the data. To prevent the server from "borrowing" the data elsewhere or from constructing the proof using some reduced form of the data and possibly other copies, each redundant copy *UQC* is of the form:

$$UQC = \mathsf{kpbl} \big| \{nonce \big| data\}_{\mathsf{kprv}}$$

The proof *prf* that the server still holds the data is constructed as follows:

$$prf = \| C\_Auth \big| UQC \| ; S\_Auth$$

Here the C_Auth is a random string constructed by the *data* creator. It prevents the server from pre-computing the proof. The next part of the proof, the S_Auth, is constructed by the server and prevents the data creator from stealing the fees. Successful recipient of fees has to construct both parts of the proof.

The bank which processes the money transfer checks the correctness of the resulting *prf*.

Bank credits money received from a payer to the common Pursue Account and stores only a collection of strings which indicates when and under which proof it should transfer the same sum to a requester. First successful proof constructor takes the whole fee. There is no indication who it should be nor any connection to any piece of data. There is no way to match incoming a payment with the corresponding outgoing one.

Let $\mathcal{C}$ be a client, $\mathcal{E}$ be an Eternity Server and let $\mathcal{B}$ be a bank. For sake of simplicity we assume only one payment that will be carried out after the storage period has expired. The whole protocol looks as follows[6]:

1. $\mathcal{C} \longrightarrow \mathcal{E}: \; Time; Size; T\_id; \bigvee_{i=1}^{n} (\overbrace{\{T\_id | Probe | seed_i | dgst_i\}_{\mathsf{ks}_i}}^{C\_Auth_i})$
2. $\mathcal{E} \longrightarrow \mathcal{C}: \; \begin{cases} Refuse \\ T\_id; S\_id; index; (Bank; Value; Bound; Exp); \{S\_id | S\_Auth\}_{\mathsf{Kpbl}_{\mathcal{B}}} \end{cases}$
3. $\mathcal{C} \longrightarrow \mathcal{B}: \; [Value | Bound | Exp]_{\mathsf{Kpbl}_{\mathcal{B}}}; \bigvee_{i=1}^{n} ([C\_Auth_i]_{\mathsf{Kpbl}_{\mathcal{B}}}); [S\_Auth]_{\mathsf{Kpbl}_{\mathcal{B}}}; \wr \wr \bigvee_{i=1}^{n} ([dgst_i]_{\mathsf{Kpbl}_{\mathcal{B}}}; [seed_i]_{\mathsf{Kpbl}_{\mathcal{B}}}); S\_id; T\_id$
4. $\mathcal{B} \longrightarrow \mathcal{C}: \; \langle Value | Bound | Exp | T\_id | S\_id | B\_id \rangle_{\mathsf{Kprvs}_{\mathcal{B}}}; \wr \bigvee_{i=1}^{n} (\langle c\_auth_i | T\_id | S\_id | B\_id \rangle_{\mathsf{Kprvs}_{\mathcal{B}}}); \langle SA | T\_id | S\_id | B\_id \rangle_{\mathsf{Kprvs}_{\mathcal{B}}}$
5. $\mathcal{C} \longrightarrow \mathcal{E}: \langle Value | Bound | Exp | T\_id | S\_id | B\_id \rangle_{\mathsf{Kprvs}_{\mathcal{B}}}; \bigvee_{i \neq index} (\mathsf{ks}_i); UQC; \wr \wr \bigvee_{i=1}^{n} (\langle c\_auth_i | T\_id | S\_id | B\_id \rangle_{\mathsf{Kprvs}_{\mathcal{B}}}); \langle SA | T\_id | S\_id | B\_id \rangle_{\mathsf{Kprvs}_{\mathcal{B}}}$
6. $\mathcal{E} \longrightarrow \mathcal{B}: \; B\_id; S\_id2$
7. $\mathcal{B} \longrightarrow \mathcal{E}: \; B\_id; S\_id2; \begin{cases} \bigvee_{i=1}^{n} (seed_i) \\ Err\_indication \end{cases}$
8. $\mathcal{E} \longrightarrow \mathcal{B}: \; B\_id; S\_id2; \bigvee_{i=1}^{n} (dgst_i)$

[6] $\langle m \rangle_{\mathsf{kprvs}}$ denotes digital signature of message m with private signing key kprvs—note that it is the signature itself, it does not include the signed data.

9. $\mathcal{B} \longrightarrow \mathcal{E}$: $S\_id2; \begin{cases} Success \\ Fault \end{cases}$

Here *Time* denotes the requested period of data deposition, *Size* stands for the size of the stored data. *Value* is the price of storage. *Bound* characterises the time when server can request fees and *Exp* is the time when the client can request fee refund. *S_id*, *T_id*, *B_id* are identifications assigned to the transaction by the server, the client and the bank respectively. $\mathsf{Kpbl}_{\mathcal{B}}$ is the public key of the bank involved while $\mathsf{Kprvs}_{\mathcal{B}}$ is the signing key of the bank.

When we allow payment reimbursement we have to preclude client from constructing the pair *seed*;*proof* so that the server will be unable to obtain fees. This is why we use n pairs simultaneously. The server can check $n-1$ pairs before it starts to provide service. The missing pair makes the server to keep the whole data. Note that the server in step 1 obtains all pairs enciphered and in the following step it selects at random the *blind key*, i.e. the number (here denoted by *index*) of the key, which the client does not provide to the server in step 5.

4.5 Distributed Name-space and Data Retrieval

When a client stores some information in the Service, he associates a list of keywords with it. These keywords should characterise the information and enable people to easily retrieve the information later. The server upon receiving the data computes a value called *internal checksum (ICK)*:

$$ICK = \|data\|$$

Because *ICK* is the result of a hash function simply applied to the original data (not the UQC!), internal checksums of all redundant data copies representing the same information are exactly the same. The server associates the *ICK* with data as another keyword.

This way we achieve a unique (with high probability) name-space without any need of a central policy agent. Furthermore, this model is quite resistant to attackers who try to substitute different information.

Client who wishes to retrieve some information has to perform these steps:

1. By issuing (several) Find requests, he obtains the *ICK* of the requested information.
2. Having the *ICK*, the client can issue a Data Request to obtain the requested piece of information.

To manage the first step, the client has to put together a characteristic of the required information. He sends the characteristic to an arbitrary Eternity Server. The characteristic could be a regular expression or a similar tool allowing searching the name-space (i.e. keyword-lists). Each server that obtains such a request follows this algorithm:

1. The server checks own data structures and puts together a list of keyword-lists best matching the obtained characteristic.

2. It updates the depth of the search according to $depth = \min(depth - 1, max\_depth)$
3. If $depth > 0$, it selects at random several other servers and submits the request to them.
4. It collects the responses, removes the redundant ones and sends the result back to the immediate requester.

Each record in the resulting list contains *ICK*. The client checks the result if there is the desired information listed. If so, he has its *ICK* and can issue a Data Request, otherwise he has to re-arrange his request and repeat the query.

After reception of a Data Request, Eternity Servers perform similarly shaped search as in the case of Find Request. When a server locates requested data either within its own data structures or in a response from another server, it passes the data to the requester and stops processing the request. This way, the Service delivers at most one copy of the requested data to client.

4.6 Protection of Stored Data

The stored data is uniquely identified to clients by an *ICK*. Both Find Request and Data Request commands work with it. The data is protected against an external attacker by unknown location and selection of available functions. To strengthen data protection, we use secondary identification *FileId*. The server associates this server-wide identification with each piece of stored data.

The Eternity Server has three principal persistent data structures:

- stored data files—the actual data stored by clients; the data is encrypted and visibly identified (i.e. accessible by the administrator) by *FileId*.
- list of installments—here the server keeps the information about the times when it should request fees and any necessary information connected with payments; the corresponding data file is identified by *FileId*.
- Index—data structure that contains keyword lists associated with each data file, it also provides connection between these lists and both *FileId* and *ICK* identifications; the whole Index is kept encrypted and available only to the server's internal processes.

This way all administrative tasks are based on the *FileId* identification while all operations connected with data content make use of the *ICK* identification. Administrators can not easily manage data in accordance with its content.

We proposed a method how to store data so that each stored piece of data is connected with several other ones so that it is impossible to remove particular data without damaging the others. There is also a possibility of implementing all critical parts of server functionality in hardware TCB. Description of a possible interface is beyond the scope of this article.

4.7 Long-term Pseudonyms in Anonymous Environment

To allow for repeated requests to the same server under ERM we introduce digital *pseudonym* for each Eternity Server. The pseudonym is continuous, i.e.

server can keep it for a long time period within which several re-establishing of the pseudonyme and number of public key changes can occur. It does not reveal any useful information about the owner's real identity and can not be easily forged.

We use an external source: a regular Certification Authority (CA). To bind the server's public key with its digital pseudonym we introduce a new type of certificate—an *Identity Certificate (IC)*. We achieve greater reliability and survivability by employing several authorities in the process of IC creation. Each CA has to ensure that it will not issue a certificate bound to the same pseudonym twice.

Our primary interest is to establish continuous identity (the pseudonyme) to which keying material is bound as usually. The certificate owner has to obtain certification from a specified number of previously involved CA-s to achieve a valid IC that could be accepted as a successor. The owner has to compensate an unavailable CA with another one.

Identity certificate contains the following information:

- *Dig_pnyme*—digital pseudonym of the entity, selected at random. It should be long enough to ensure uniqueness (recall that it is selected without coordination with rest of the world).
- validity—a usual not_before–not_after pair
- $\bigvee_1^m(duration)$—how long each certification authority maintains continuity of this pseudonym (i.e. date of issue of the first predecessor)
- Kpbl—the public key belonging to the certificate
- $\bigvee_1^k(Rev\_URL)$—a list of URL-s where revocation can be checked
- $\bigvee_1^l(Prev\_Crt)$—a list of digests of one or more previous certificates
- $\bigvee_1^m(CA\_id)$—a list of identification of certification authorities willing to sign the certificate
- $\bigvee_1^n(CA\_sig)$—actual signatures created by certification authorities listed in the previous list; each signature is of the form:

$$\overbrace{\underbrace{\langle Dig\_pnyme | validity | \ldots | \bigvee_1^l(Prev\_Crt)}_{id\_data} | \bigvee_1^m(CA\_id)\rangle_{\mathsf{Kprvs}_{\mathcal{CA}}}}^{id\_info}$$

To establish a digital pseudonym, the owner performs the following protocol with each involved certification authority:

$$1.\ \mathcal{E} \longrightarrow \mathcal{A} : \{id\_data; old\_ticket\}_{\mathsf{Kpbl}_{\mathcal{A}}}$$

$$2.\ \mathcal{A} \longrightarrow \mathcal{E} : \begin{cases} \langle \| id\_data \| \rangle_{\mathsf{Kprvs}_{\mathcal{A}}} \\ Refuse \end{cases}$$

$$3.\ \mathcal{E} \longrightarrow \mathcal{A} : [id\_info; old\_ticket; \mathsf{k}]_{\mathsf{Kpbl}_{\mathcal{A}}}$$

$$4.\ \mathcal{A} \longrightarrow \mathcal{E} : \{\langle id\_info \rangle_{\mathsf{Kprvs}_{\mathcal{A}}}; new\_ticket\}_{\mathsf{k}}$$

4.8 Time-keeping

Proper time-keeping is essential. Moving the time forward may make servers discard stored data prematurely. Such situation can endanger the whole system of payments. Mixes with wrong internal time will refuse valid messages which may subvert the whole ERM.

It seems that the best solution is to use an external source of reliable time synchronisation. Eternity Servers should use the full NTP protocol [8] and several different sources of precise time information. Other servers can employ regular SNTP [7] client functionality.

The Eternity Server should not delete any data for which it is unable to obtain seeds from the bank in step 7 of the payment protocol (see 4.4).

5 Remarks About the Service Servers

Our main interest is to keep all servers running under any circumstances. Thus if a server encounters any error, it simply aborts the current computation, discards all temporary data and prepares itself for processing a new request. The server also measures the time spent in the processing of each task. If this time exceeds a defined value, the server immediately finishes the operation and cleans-up all associated data. The server processes any operation independently from all others.

6 Results

6.1 General Observations

There is no way to protect information reliably. An attacker can effectively gain any access the legitimate users have. This conclusion implies that it is particularly difficult to ensure information secrecy.

Once particular information of an attacker's interest is located, there is little the system can do to protect it against damage or destruction. A good perimetric protection can resist both professional and amateur opponents attempts to some degree but a determined attacker can overcome it. The same holds for appropriate backup. The influence of authorities can be reduced somewhat by suitable techniques of data storage.

Surprisingly, no custom hardware, such as various trusted computing bases, can bring a significant contribution to overall system reliability or immunity to some attacks. Such hardware can, to a certain limited extent, strengthen resistance of individual servers but definitely can not strengthen cooperation with other servers. Disadvantages of incorporating such hardware outweigh benefits.

Cryptographic techniques provide only a reduction of the possibly large sensitive data to a significantly smaller key. The cryptography itself can not ensure information protection under our threat model. The value of cryptography lies primarily in the fact that it can extend the attack price substantially.

6.2 Data Survivability

We assume that the proposed mechanism provides good level of protection against information transfer tracking. An attacker who wants to track datagrams has to assault and analyse all Mixes along the transfer path; furthermore he must do this task "on-the-fly" because the Mixes tightly adhere to forward secrecy principles. Eternity servers can issue Access Certificates with reasonably short validity period which prevents the attacker from accumulating of knowledge about their location. It seems very difficult to perform a selective attack directed against the servers that store particular information.

Fig. 1. left: An estimate of information survivability. right: Probability that located data will be successfully delivered the to requester.

6.3 Service Reliability

Recall how Eternity Servers co-operate in the searching or retrieval of some piece of information. Because of a lack of global coordination, they can reach any data only with certain probability. Thus the responses from the Service to exactly the same request may differ from time to time.

6.4 Drawbacks

Unfortunately, the mix-network does not prove to be particularly suitable data transport mechanism. Even though the long line of consequent Mixes is strengthened by redundancy at each node, it offers only mediocre resistance to a massive attack. Although this does not endanger the stored information itself, it can make access to it slow or otherwise complicated at least temporarily.

7 Ethical and Legal Consequences

Service features have strong impact to contemporary view of intellectual property rights. Just imagine that somebody stores an MP3 file with a popular song. Also the need to restrict the administrator's ability to control the stored data according to its content may not be considered ethical.

8 Conclusion

We presented a complete description of a system based on the ideas of Anderson's Eternity Service. Our goal was to strengthen the resistance of the system against attacks to achieve as good data survivability and availability as

possible. Proposed payment support preserves recipient anonymity and is relatively robust. Our system of information identification (ICK in conjunction with a keyword-list) represents a name-space that is resistant against data substitutions while easily usable by humans. We developed a system of permanent pseudonyms allowing reliable use of digital signatures within a totally anonymous environment. An important part of our work is a mission-independent mutually anonymous message transfer mechanism capable of supporting other systems requiring anonymity. The Service is almost immune to brute force. Until the exact location of data storage becomes known, our adversary has only little chance of destroying the corresponding information.

References

[1] Ross J. Anderson. The eternity service. In *Pragocrypt 1996*, 1996. `http://www.cl.cam.ac.uk/rja/eternity.html`.

[2] A. Back. The eternity service. *Phrack Magazine*, 7(51), Sep 1997. `http://www.cypherspace.org/~adam/eternity/phrack.html`.

[3] I. Brown. Eternity service design. `http://www.cypherspace.org/eternity-design.html`.

[4] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, Feb 1981.

[5] Wei Dai. PipeNet 1.1. `http://www.eskimo.com/~weidai/pipenet.txt`.

[6] D. M. Goldschlag, G. R. Michael, and P. F. Syverson. Hiding routing information. In *Workshop on Information Hiding*, Cambridge, UK, May 1996.

[7] D. Mills. Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI. Technical Report RFC-867, University of Delaware, Network Working Group, Oct 1996.

[8] David L. Mills. Network time protocol (version 3) specification, implementation and analysis. Technical Report RFC 1305, University of Delaware, Mar 1992.

[9] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communication Special Issue on Copyright and Privacy Protection*, 1998.

[10] Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. Anonymous connections and onion routing. In *18th Annual Symposium on Security and Privacy*, pages 44–54. IEEE CS Press, May 1997.