

# Pesto Flavoured Security

Feike W. Dillema and Tage Stabell-Kulø  
Department of Computer Science, University of Tromsø, Norway  
{feico,tage}@pasta.cs.uit.no

## Abstract

*We demonstrate that symmetric-key cryptography can be used for both read and write access control. One-time write access can be granted by handing over an encryption key, and our encryption framework allows the revocation of previously granted rights. The number of keys to be managed explicitly grows linearly with the number of access control policies a user defines, making security manageable.*

*The framework is used in the Pesto distributed storage system. In Pesto, policies can be stored the same as other data and the same mechanism can be used to control access to them. Delegation of authority over policies concerning different tasks can then be performed. Separating the different tasks of the system, allows for different tasks to be assigned to different sets of nodes. Nodes need then only be trusted wrt. the specific task(s) they have been assigned with.*

## 1. Introduction

We aim at providing an infrastructure that meets the needs of mobile users for distributed, reliable and secure storage. We believe that the crux of the matter is related to trust, and management of trust relations. An important part of Pesto is to separate trust relations from administrative relations, and to separate availability of *storage resources* from accessibility of *content*.

We believe that for a system to be viable in a rapidly diversifying world, the overall system design cannot rely on a common policy. Not a common policy for authentication, not for authorization, and not for the cost and value of resources. Furthermore, we believe that the users should be provided with the means to establish any relationship to service providers as they wish. The system should utilize the relationships the user might have, instead of requiring the establishment of specific relationships from the user a priori.

Mobile machines and users require fundamentally different solutions to availability, safety, and privacy. At times,

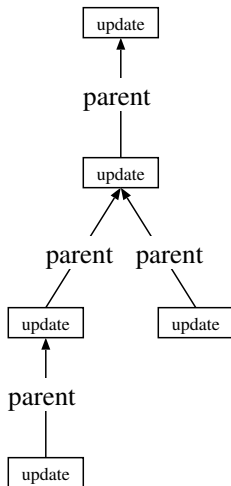
users lack (good) connectivity to the whole or part of the infrastructure. Most existing systems focus on efficient use of local storage resources for availability during such periods. We assume, however, that storage resources are plentiful in many environments a user may roam into. The resource that is scarce in these environments is trust, while access to more trusted parts of the infrastructure may be severely limited.

Our focus is on efficient use of the trusted resources available to a user in order to reduce the security and safety risks of using untrusted resources in his current environment. Users may have different levels of trust in different parts of the infrastructure. He may, for example, merely trust a node to store (encrypted) data, and/or to distribute replicas to other nodes on his behalf, and/or he may trust a node to enforce access control on his behalf to his (plaintext) content.

We have designed and implemented the Pesto distributed storage system. It's intended use is as bottom layer of a full-fledged distributed filesystem (but is certainly not limited to that). As such it is meant to implement basic mechanisms for secure storage and replication of data, together with minimal support needed by higher layers to implement more advanced functions like locking schemes, consistency control, hierarchical namespace(s), location services, service contract negotiation and authentication of users. The Pesto storage system stores data and provides basic mechanisms for implementing replication and access control policies.

The Pesto system model consists of nodes that communicate with each other using an asynchronous request-response protocol that supports two types of requests; one to fetch data from a node, and one to store data at a node. A node is owned by a user who has authority over its local storage resources. A user may acquire the right to use storage resources at a remote node by means of a service contract. A user delegates such rights to Pesto such that it can use these to implement the user's (replication) policies.

Encryption of a file moves its security requirements from that file to a key. Managing the security requirements for (small, fixed-length, relatively long-lived) keys is typically a much simpler problem than managing those of files con-



**Figure 1. A file**

taining arbitrary data. The Pesto encryption framework makes use of multiple levels of encryption to minimize the amount of data and number of keys that need to be managed explicitly by higher levels or end-user.

All data stored in Pesto is encrypted and access to files is reduced to access to keys. Access to storage space then means fetching and storing encrypted data at a node, while access to content means access to the encryption key that is needed to decrypt that data. Pesto allows nodes to share storage resources independently from actual content.

The design of Pesto carefully separates the different mechanisms a distributed storage infrastructure must support. A user can then place responsibility of the different tasks on different sets of machines, possibly governed by a variety of administrative domains.

## 2. Files

The Pesto storage system provides distributed storage of files to its users. Pesto stores and replicates the complete update history of files. More precisely, a file in Pesto is an initial (empty) version, together with all updates made to that file. As such, any version of a file can be retrieved at any time. Pesto puts no constraints on the content of a file update at all, and leaves it to applications to define their own update and access semantics. For example, an application may choose to store the actual differences between the current version and a new one, or it may store the complete new content as a file update. The Pesto ‘current version’ of a file is the current set of updates of the file, i.e. its complete update history. It is up to applications using Pesto to present the user with an up-to-date and consistent view of the file’s content and define the ‘current content’ of a file.

A file update is immutable and is identified by a *globally unique identifier* (GUID) which is a 128-bit random number. By convention, a file is identified by the GUID of its (empty) initial update. Due to the random selection from an immense name space, a new GUID can be generated locally with very little risk of an identical name existing anywhere in the system. A file update keeps a reference to the previous update to the file, its parent update. The parent reference, i.e. the parent’s GUID, is stored and distributed with the file update. The root (initial) update of the file has no parent and keeps no such parent reference. A file consists therefore of an ordered set of identifiable updates. As multiple updates may refer to the same parent, a file is organized as a file update tree. An example file is shown in Fig. 1.

A Pesto node maintains a special file that contains a variety of administrative information about it. This file describes and represents the node, its owner and its storage resources. Its GUID is used to identify and address the node in the system.

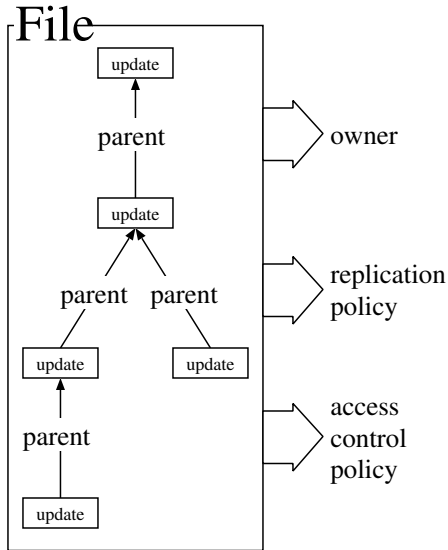
A file update has a *creator* associated with it. The creator of an update is the (GUID of the) node that authorized its creation. The creator of the root (initial) update of a file is special, as explained in the next section, and is called the *owner* of the file. The GUID of the creator is kept with the file update and is immutable. This means that ownership of a file is not transferable from one user to another, other than by making a copy of the file that has a new GUID.

A replication policy and an access control policy is associated with a file. These policies are also stored as files. A file references a policy that applies to it, by the GUID of the file that stores the policy. These policy references are specified by the user at file creation time and are immutable. A given file can thus never be associated with a policy identified with another GUID. The content of the policy files themselves can be changed, of course.

A replication policy specifies the set of nodes where the user expects to store a replica of his file, and it also specifies the nodes responsible for the distribution of the replicas to these nodes.

Distribution of replicas according to user specified policies proceeds whenever communication between source and destination node is possible, i.e. is independent of any synchronization with other replicating nodes. As there is no mechanism (protocol request) for deleting individual files or file updates by GUID, the only way to ask a node to remove files from its local storage is by removing that node from a replication policy. Note, however, that such a request will apply to all files that are governed by that replication policy.

The combination of replication and independent, concurrent updates requires that the issue of consistency must be considered. In general, it is impossible to ensure that no concurrent updates occur unless one is willing to accept that an update blocks until the global state of the system can



**Figure 2. A file stores the GUIDs of its owner, a replication and an access control policy**

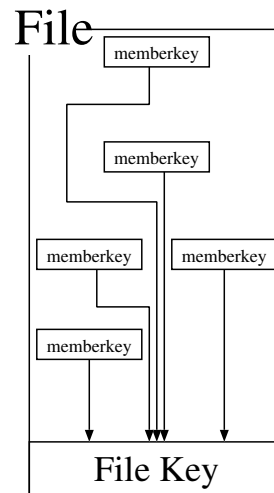
be determined. However, since each update in Pesto has an unique name, the two (or more) concurrent updates are identifiable. Because they are concurrent, they will have the same parent, i.e. they are derived from the same version. It is left to each and every application, and ultimately to the user, how to deal with a concurrent update (a branch in the update tree). This design-choice implies that the term replication in the context of Pesto merely refers to the activity of distributing copies of file updates to a user-specified set of nodes. In particular, it does *not* include distributed consistency control. Pesto thus separates replication from consistency control, and the storage system itself only provides replication; the advantage is a significant simplification in the design.

An access control policy specifies what credentials a user deems sufficient for a request to be granted access to the content encoded by his file, i.e. who is allowed access to its updates. As the storage resources of a node are described and represented by a file, a regular access control policy can be used to specify who should be granted access to these storage resources. In other words, access to content and storage resources are separated, but governed by the same mechanisms.

In short, we can say that Pesto stores and distributes the complete update history of content in files, and it stores all state of the system in such files. References by GUID are used to associate files and policies with each other.

### 3. Security

Pesto itself is designed around a very simple base security policy: all communicated and stored content is regarded confidential and access is only granted to the user that owns the content. A file is (and stays) owned by the user that initially creates it, and the encryption keys needed to decrypt it are initially only available to the owner of the file. A user may relax this base security policy for the files he creates by specifying what other Pesto nodes should be granted access to the various encryption keys in use. Pesto is responsible for securely distributing the encryption keys to nodes that the user trusts.



**Figure 3. File encryptions**

#### 3.1. File Encryption

The complete update history tree is stored for every file as a set of *file updates*. The content of each file update is encrypted with a different encryption key. A key used to encrypt a single file update is called a *member key*. Read-access control is then exercised by controlling who has access to a file's member keys. The member keys of a file are subsequently encrypted with the so-called *file key*. The example file depicted in Fig. 1 thus uses five member keys and a single file key as shown in Fig. 3.

Each encrypted member key is stored as part of the file update it belongs to. The file key is generated when the file is created, at which time it is made available to the owner of the file. A user who knows the file key is able to get hold of all member keys, which gives him read access to all file updates (i.e. the full update history of the file). Obviously, by handing out a single member key, read access is granted for individual file updates.

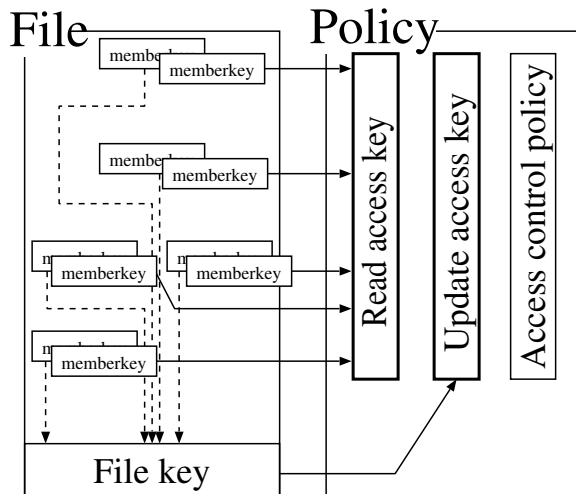


Figure 4. Access control policy

### 3.2. Update Authorization

Update access to files is controlled by the same mechanism used to control read access. When a request to update a file is received in the form of a new file update, it is considered authorized if the update is encrypted with a fresh member key, and that key is found encrypted with the file key for that file. This means that update access can only be granted by someone that knows the file key.

A member key is deemed fresh when no (locally) existing file update of its file is encrypted with that member key. A node that knows the file key can check the encryption of the file update and the freshness of the member key used, and decide whether the update is properly authorized.

Anyone who controls the file, i.e. has access to the file key, can properly encrypt a new member key and hand it to any user (including himself). This constitutes a delegation of authority over the file for the purpose of a single update. Notice how this update authorization mechanism ensures that a user can be given authority to update a file one single time, without him learning the file key. Hence, the authority is not reusable and has one-time use semantics.

The following example featuring the users Alice and Bob may clarify how this mechanism works. Alice creates a file resulting in an initial file update encrypted with a newly generated member key. This member key is encrypted with a newly generated file key (and stored and distributed with the initial file update). Alice knows the file key and keeps it secret. In order to make a new update to the file, Alice needs to generate a new member key to encrypt the file update with, and encrypt this member key with the file key.

Later, Alice decides to work together with Bob for a while and let him make a new version of her file. In order to

authorize Bob to make an update to her file, she generates a GUID and a new fresh member key for the update Bob is planning to make. She stores and distributes the member key (with the GUID) encrypted with the file key and hands the plaintext member key and GUID over to Bob over some secure channel. Note that Pesto is not concerned with how Alice and Bob identify and authenticate each other, nor what the initial secure channel between them looks like. Some time later, Bob can then encrypt his update in a way that will convince Alice (or any other with access to the file key) that the update is a properly authorized one.

### 3.3. Revocation

As a consequence of how authority over updates is implemented, authorization to make a file update can be separated in time from the actual injection of the update into the storage system. There may then sometimes be a need to revoke this authorization after it has been granted but before it is used. Imagine, for example, situations where users hoard authorizations or try to save authorizations for use long after the original credentials used to acquire them expired. A user may find it useful or required to avoid such situations. In order to revoke authorizations, the user that issued the authorization can store an ‘empty’ file update with the GUID and member key in question himself. Due to the WORM (Write Once Read Many times) access semantics of file updates stored by Pesto, this effectively renders the update authorization harmless and so revokes it.

One could argue that revocation of read access to existing updates is useless as it is impossible to make a user forget what he just read (or copied). But accidents happen occasionally and one wants to be able to clean up after oneself as much as possible before the full dreadful consequences of a mistake or security compromise realize themselves. We consider revocation of read access rights, however, to be a relatively rare recovery operation that does not warrant explicit support at the cost of increased system complexity. Revoking read access by revoking (changing) member keys is therefore not supported. In order to revoke read access one therefore has to copy the relevant file content to a new file and stop replicating the old one. The WORM access semantics then not only apply to the plaintext of updates but also to encrypted file updates.

### 3.4. Access Control Policy

We described in Sec. 2 how each file references an access control policy by the GUID of the file that stores it. Such an association between a file and its access control policy needs to be protected by cryptographic means in order to be useful. To that end, an access control policy file contains two encryption keys. These keys are called the *read access*

key and the *update access key* respectively. The read access key of an access control policy is used to encrypt the member keys of the files that are governed by the policy. The update access key of an access control policy is used to encrypt the file key of each file that references the policy.

A member key encrypted with the read access key is stored and distributed with its file update. A file key encrypted with the update access key is stored and distributed with its file. The example file depicted in Fig. 1 thus stores five member keys encrypted with the file key, five member keys encrypted with the read access key, and one file key encrypted with the update access key of its access control policy as shown in Fig. 4.

Basically, we group files under their respective access control policy. We avoid storing and distributing a potentially very large number of member keys and file keys with the policy. This is achieved by adding an extra level of cryptographic indirection, i.e. two new keys that encrypt those member keys and file keys instead. The result is that the (encrypted) member keys and file keys can simply be replicated together with the files, to increase safety and availability.

Of course, the requirements for secrecy, safety and availability have moved from the member keys and file keys to the access keys, i.e. we still need to keep these two keys safe and secure. However, this is a much simpler problem to solve. Not only are there far fewer keys to protect, but more important is that the qualities of these two keys are substantially different from those of the set of member and file keys. In particular, while the set of member and file keys will grow over time as new files are created, the set of two access keys does not change with the number of files they apply to.

As only nodes that hand out member keys to users need to know the access keys, it will typically be easier to maintain their secrecy. Safety is also easier to maintain due to the small amount of data involved and because this data is immutable. It will typically be feasible to replicate the access keys to physically secure and safe (offline) media, like a smartcard kept in a safety deposit box of a bank. To achieve availability, we expect the mobile user to carry his access keys with him on a smartcard or small handheld computer, optionally protected with a passphrase in addition. Hence, secrecy is achieved without complicating the implementation of safety and availability requirements.

### 3.5. Granularity of Control

The encryption framework presented so far achieves that control can be exercised at fine granularity. Both read and update access is reduced to access to member keys (respectively to existing and newly created member keys), and the unit of access is the *file update*. As described in Sec. 2, there are no constraints on what the content of a file update actu-

ally constitutes. It is left to the application to specify file update semantics. This also means that applications can determine the unit and granularity of storage, replication and access control.

A straightforward example is a versioning file system application, that stores each file version as a Pesto file update, such that the unit of access control is a file version and access to different versions can be controlled independently of each other. Another possibility would be to store the differences between the previous version and the new version as a Pesto file update to improve storage efficiency at the cost of simplicity.

Pesto itself defines somewhat less conventional file update semantics for its administrative and policy files with as main goal simplicity. After all, little is gained by leaving functionality, like consistency control, to be implemented by higher layers for user data, if much of the same functionality needs to be implemented for system data anyway.

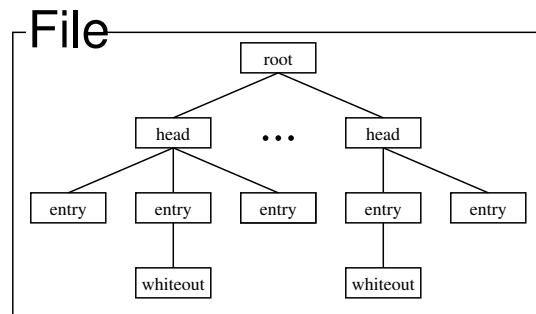


Figure 5. Policy file structure

Policy files are, for this reason, organized as one or more unordered lists as depicted by Figure 5. A replication policy file includes the user-specified lists of replicating nodes, and an access control policy file includes access control lists. Remaining content, like the access keys for the access control policy file, is stored at the heads of the lists.

The number of lists (and thus the head file updates) is fixed but list entries can be added and removed. A list entry is stored as a file update that has a head file update as parent. Removal of a list entry is done by means of a so-called *whiteout update* that disables its parent list entry. Both adding and removal of a list entry are then idempotent operations, and any valid update to a policy file at a node brings the file from one consistent state to another.

As the list entry updates are assumed to be independent from one another, distributed consistency control reduces to a quality of service problem for the distribution of the list entry updates. Under the assumption of fair links between nodes, all updates will eventually arrive at all nodes and result in the same state at all nodes, i.e. distributed con-

sistency will be achieved eventually. A Pesto node does not need to know *when* global consistency is achieved in order to continue work locally, and therefore does not need further consistency control mechanisms.

## 4. Tasks and Trust

The design of Pesto carefully separates the different mechanisms a distributed storage infrastructure must support. The trust relation a user has with the (owner/manager of the) node will determine how the user will use the node in his Pesto configuration. Note that there is really no such thing as a “Pesto system”. There is merely a collection of completely independent Pesto nodes, and each and every user determines to what extent he trusts particular nodes of his choosing.

A node can be assigned the responsibilities for the tasks of storing files, distributing files, and/or enforcing access control to files on behalf of the user. The manner in which encryption is applied reduces the assignment of a role to a node to the exchange of an encryption key. An assignment of responsibility is limited to a user-specified set of files, i.e. different nodes may be assigned responsibilities regarding different sets of files. As a result, a user can specify three different sets of nodes for each of his files. These sets are called the trusted storage base, the trusted replicator base, and the trusted access base.

### 4.1. Trusted Storage Base

The trusted storage base (TSB) of a file is the set of nodes the user trusts to store an (encrypted) replica of that file. A member of a TSB is only trusted to store data, it is not trusted with the keys that protect the data. Nodes in a TSB are thus ‘merely’ storage providers for the user and they perform access control to their storage resources, not of the content they store.

Users negotiate service contracts with other users in any way they deem appropriate. We envision that users establish service contracts with other users in a variety of ways; online storage service providers offering services for a fee, cooperative users exchange storage for storage, and companies might offer their employees access to its storage resources. A service contract could take many different forms, from paper contracts to electronic contracts or just verbal agreements. How such contracts are established is of no concern to Pesto, regardless of the relationship and conditions of use.

The only requirement on a service contract negotiation is that somehow a secret key is exchanged as part of it. This secret is subsequently used to authorize requests to use the negotiated storage resources. A request to use the negotiated storage resources, i.e. store a file update, is consid-

ered authorized if the request is properly encrypted with this shared key, and accounting by the storage provider shows enough negotiated storage resources are still unused by the user.

If a user wants to act as storage provider, he does so by specifying an access control policy for his storage resources. Access control of storage resources at a node is reduced to access control of the file that describes these resources. This reduces the complexity of the storage system.

### 4.2. Trusted Replicator Base

The trusted replicator base (TRB) of a file are the nodes trusted to enforce its replication policy on behalf of the user; that is, each TRB member is responsible for the *distribution of replicas* to some subset of nodes in the trusted storage base of the file. Together with the TSB, the nodes in a TRB make up a directed distribution graph with edges leaving only nodes in the TRB and ending in nodes in the TSB. A user shares a secret key with each of his replicators, like he does with his storage providing nodes.

In order to perform its assigned task, a member of a TRB needs authority to use storage resources negotiated by the user at the nodes it is expected to distribute file updates to. A straightforward implementation would support this by handing the secret key shared between the user and the storage provider to the replicator node. In Pesto, however, we delegate authority from this key to a new key which is subsequently installed both at the storage provider and the replicator. This facilitates easy revocation of such a delegation when a user removes an individual node from a TRB.

### 4.3. Trusted Access Base

The trusted access base (TAB) of a file are the nodes that are trusted to enforce an access control policy on behalf of the user. Two separate TABs can be specified for a file; one for read-only access, one for read/update access. Members of the former are trusted to handout read access only, and are handed the read access key in order to enable them to do so. The latter is trusted to perform both read and update access control and is handed the update access key associated with the access control policy in question.

Pesto itself is not responsible for enforcing the user-specified access control policies, and thus does not prescribe its contents. The user (with help of his management applications) specifies what (type of) credentials he finds necessary and sufficient to authorize access of some kind. The storage system is merely responsible for distributing such policies, together with the keys required to enforce them, to the relevant trusted access bases.

## 5. Related Work

The Echo file system [2] of Taos [16] relies on a system-wide trusted computing base [10, 11]. The Bayou [15], Coda [8] and Ficus [5] distributed file systems target mobile clients in the system, and use replication to improve availability while weakening consistency and introducing specialized conflict resolution schemes. These systems focus on efficient and transparent hoarding/caching for availability of data during disconnected operation.

The persistent storage architecture OceanStore [9] relies on public-key cryptography for update integrity checking, where Pesto does not. OceanStore defines a class of servers that are trusted to act on behalf of its users, where Pesto lets the user specify these.

Survivable storage systems research, like Pasis, shows how decentralization, history logging and threshold schemes can increase availability and integrity under the assumption that storage servers and their clients can be compromised [17].

SPKI provides for an elegant and simple, but yet flexible mechanism to express authorization [3, 4]. Like Pesto, SPKI takes a decentralized approach. Neither Pesto nor SPKI has any system, global or centralized notion of a trusted computing base. Similarly, Snowflake provides end-to-end naming and authorization across administrative boundaries [7]. It is not as “key-centric” as SPKI and allows for other principals than public keys only (like local channels, shared key secure channels) as Pesto does.

## 6. Discussion

Pesto has been designed around our so-called ‘*Open-End Argument*’ design guideline [14]. According to it, *the user* should be solely in charge of important matters such as how identity of users is represented and checked and what kind of credentials are needed to authorize actions. Pesto does not support any notion of authentication of “users”, and does not prescribe the structure and organization of the user community.

Our design has a user-centric view in that all authority in the system originates from individual users and not from inside the system itself. The access control and replication mechanisms do not mandate any hierarchical, fixed or static structure on administrative domains. This makes Pesto suitable for building personal ad-hoc infrastructures for sharing between individuals, but individual administrative domains can also be used as building blocks to construct larger domains using delegation. This requires cooperation from the individual users, as they cannot be forced by the system to delegate their authority to others.

We believe that the lack of mandatory transfer of authority is not a weakness in our design, but reflects that in most

real-world environments, user cooperation is a requirement to enforce a shared policy. Cooperation from non-malicious users may possibly be ‘bought’ instead, by making the use of shared resources conditional to such cooperation. For example, a company could define a storage policy for its file servers that allows only storage of files of employees, for which it has been delegated the rights to define the TAB, TRB and/or the TSB. In other words, the file servers will only store files that reference policies owned by the company. It could, in addition, set up its communication infrastructure such that only the servers under its control may be reached through it.

### 6.1. Responsibility and Risk Management

Pesto keeps different responsibilities separate, so that the user can allocate them to different, but possibly overlapping, parts of the infrastructure. Other responsibilities than presented here could be defined. For example, consistency control policies and ‘trusted consistency bases’ could be defined. As consistency requirements are highly application dependent, Pesto only provides a basic synchronization mechanism that applications can use to construct their own consistency protocols. It is outside the scope of this paper to discuss consistency control and ways to enhance the security of such protocols (see e.g. [6, 13]).

As described, authority is delegated from the user to an encryption key. This reduces the assignment of responsibilities to a key management problem. By the *scope* of a key, we mean the information that is available to a user that knows the key. For example, the scope of a member key is thus a single file update, and it has a very limited scope compared to an update access key. An important aspect of our encryption framework is that each key has a well-defined and limited scope. This assists the user in assessing the risks of using potentially untrustworthy machines in order to make progress. Pesto is not designed to deny its users service in cases where a user deems it more important to continue working than to protect the confidentiality of that work. Instead, Pesto is designed to limit the risk involved in such actions.

### 6.2. Denial of Service

Our design is well suited for the construction of publication infrastructures similar to “The Eternity Service” [1] and “The XenoService” [18], that are quite resilient to denial of service attacks.

Denial of service is targeted at destroying a certain resource or at exhausting the resources of the service providers. Replication of files together with logging of all file updates assists in preventing the former. Resilience to resource exhaustion attacks can be gained by protecting re-

source use and/or by ensuring more resources are available during the attack than an attacker is able to consume. Resource protection is supported by separating access to storage from access to content and the asynchronous nature of the protocol used. Access decision for content only need to be taken into consideration after access to storage use has been granted. When the system is flooded with requests for storage resources, one can simply ignore all these without denying service to users that were granted access to storage earlier. Graceful degradation during periods of semi or full disconnected operation limits the damage of a successful network denial of service attack.

The ability to turn secret members of a TSB into members of a public TAB with merely the exchange of a single encryption key, can be used to increase the amount of available resources dynamically during an attack to counter the resources consumption of the attack.

### 6.3. The PFS filesystem

We have implemented Pesto as part of the PFS filesystem on the NetBSD operating system. The PFS filesystem contains Pesto and a layer that maps Pesto files into the NetBSD hierarchical filesystem namespace, such that traditional tools and applications can access Pesto user data and policy files. Distribution/transport of Pesto files and requests is performed by a transport agent that accesses Pesto files through the local filesystem namespace, and uses the email system for communication.

Currently, the granularity of updates is controlled manually by the user. A user first edits a 'working copy' of a file and then manually instructs PFS to 'commit' the new version of the file as a new Pesto file update. We have implemented a consistency control protocol on top of the Pesto protocol, which the user can run in order to determine the global state of a file. We plan to implement consistency control agents that will enforce consistency regimes/policies for a user.

We use a SPKI based infrastructure 'on top of' PFS, for advanced authentication, authorization and access control [3, 4, 12]. A user performs authentication and authorization based on chains of SPKI certificates and uses SPKI certificates to construct his access control policies that are subsequently stored and distributed by Pesto. Delegation is, of course, performed by giving the application the relevant access keys that Pesto manages.

## 7. Conclusions

Pesto is a flexible distributed storage system simple enough to include resource-poor mobile devices. It allows its users to specify what part of the infrastructure is

trusted to perform each specific task on his behalf. It supports incorporation of the personal and business trust relationships into the system, while not dictating or assuming such relationships as part of the design itself. Mechanisms to increase safety and availability have been designed together with privacy protection mechanisms, providing ease of management and resilience against user error and violation of trust.

Our encryption framework makes read and update access control possible that does not rely on public-key cryptography. A user need only carry a handful of encryption keys with him on his mobile machine in order to be able to off-load work to nearby, better-connected machines. This makes Pesto suitable for networks that are semi-partitioned. Support for acquisition of authorization before actual use of it provides solid and secure support for disconnected operation.

Pesto does not rely on a system-wide or even per-user trusted computing base. Instead, individual users can specify different trusted bases for different sets of files. Furthermore, Pesto supports specifying separate trusted bases for different tasks like access control and replication. This separation avoids that security requirements for one task prevent the allocation of another task (with different or weaker security requirements) to a particular node.

The separation of access to data and content allows addressing of safety and privacy separately. The degree of replication can be increased to obtain better availability without having to consider the trustworthiness or privacy policy of new storage providers. Aiming at providing good end-to-end security and safety at the same time resulted in an elegant design were the security and safety mechanisms support each other with the addition of a minimum of complexity.

## 8. Acknowledgement

This work has been supported by the Research Council of Norway under project 152280: "IPFS: An implementation of Pesto as a file system".

## References

- [1] R. Anderson. The Eternity service. In *Proceedings of the 1st International Conference on the Theory and Applications of Cryptology*, 1996.
- [2] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, Sept. 1993.
- [3] C. M. Ellison. SPKI Requirements. RFC 2692, The Internet Society, Sept. 1999.

- [4] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, The Internet Society, Sept. 1999.
- [5] J. S. Heidemann, T. W. Page, R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experience with Ficus. In *Proceedings of the Second Workshop on the Management of Replicated Data*, Nov. 1992.
- [6] M. Herlihy and J. D. Tygar. How to make replicated data secure. In *Proceedings of Advances in Cryptology, CRYPTO '87*, number 293 in Lecture Notes in Computer Science, pages 379–391. Springer Verlag, 1988.
- [7] J. Howell and D. Kotz. End-to-end authorization. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 151–164, Oct. 2000.
- [8] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, Feb. 1992.
- [9] J. Kubiawicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ACM ASPLOS*. ACM, Nov. 2000.
- [10] B. Lampson. Protection. In *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, Mar. 1971. Reprinted in *ACM Operating Systems Review*, 8, 1, January 1974, pp. 18–24.
- [11] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, Nov. 1992.
- [12] P. H. Myrvang. An infrastructure for authentication, authorization and delegation. Cand.scient. thesis, Department of Computer Science, University of Tromsø, Norway, May 2000.
- [13] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
- [14] T. Stabell-Kulø, F. Dillema, and T. Fallmyr. The open-end argument for private computing. In H.-W. Gellersen, editor, *Proceedings of the ACM First Symposium on Handheld, Ubiquitous Computing*, number 1707 in Lecture Notes in Computer Science, pages 124–136. Springer Verlag, Oct. 1999.
- [15] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. Hauser. Managing Updates in a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 172–183, Copper Mountain Resort, Colorado, Dec. 1995.
- [16] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Trans. Comput. Syst.*, 12(1):3–32, Feb. 1994.
- [17] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliccote, and P. K. Khosla. Survivable Information Storage Systems. *IEEE Computer*, pages 61–68, Aug. 2000.
- [18] J. Yan, S. Early, and R. Anderson. The XenoService - A Distributed Defeat for Distributed Denial of Service. In *ISW 2000, IEEE Computer Society, Boston, USA*, Oct. 2000.