Security in Pesto

Feike W. Dillema and Tage Stabell-Kulø

Abstract—

Pesto aims at providing highly available and secure storage for long-lived data to mobile users roaming into (potentially) untrusted environments. This paper presents and discusses the security mechanisms and features of Pesto.

Security in Pesto encompasses the following three aspects: availability, safety, and privacy. Most existing systems focus on a subset of these aspects, or assume they can be treated independently of each other. A mechanism supporting one aspect may adversely affect another. For example, replication of data may increase availability but complicates supporting confidentiality of the data, and simply encrypting data for confidentiality may defeat the whole purpose of replication. We show how an integral approach to these aspects leads to considerable savings in overall system complexity and efficiency.

Mobile environments require fundamentally different solutions to availability, safety, and privacy. Users will suffer from temporary lack of (good) connectivity to the whole or part of the infrastructure. Most existing systems focus on efficient use of the local storage resources for availability during such periods. In Pesto, however, it is assumed that storage resources are plentiful in many environments a user may roam into. He could use such resources to supplement his limited local resources. The resource that we assume to be scarce in these environments is trust, while access to more trusted parts of the infrastructure may be severely limited at the same time. Our focus is on efficient use of the trusted resources that are available to a user. Users may specify different levels of trust in different parts of the infrastructure. In particular, nodes can be trusted to merely store (encrypted) data for a user. A user may trust a node to distribute replicas to other nodes and/or he may trust a node to enforce access control on his behalf to his (plaintext) content.

This paper presents Pesto, with a clear emphasis on its novel encryption scheme and trust management. We discuss its suitability as underlying infrastructure for feature rich distributed systems.

Keywords— **distributed storage, security, safety, mobility, replication**

I. Introduction

When users embrace mobile computing, they typically end up with multiple machines, both stationary and mobile. As a consequence, they will have a need to share their data between these machines. Mobile computing adds the requirement that users should be able to roam between many different environments. Independent of what administrative domain they roam into, they want access to their personal resources while using resources available in that domain. As the content and resources being shared may then belong to different administrative domains, there is a need to separate the mechanisms for sharing these. Furthermore, users will have the need to share data with other users. Such sharing should be possible between any two communicating users, independent of their current environment and connectivity to other parts of the infrastructure.

The Pesto system aims at providing a 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 that endeavor is to separate trust relations from administrative relations, and to separate storage and availability of *data* (encrypted content) from accessibility of *content*.

Pesto consists of nodes that communicate with each other using the P_2P protocol. The P_2P protocol is an asynchronous request-response protocol that supports two types of messages; one to fetch data from and one to store data at a remote node. All data stored in Pesto is encrypted with a shared encryption key. Access to storage space, i.e. fetching and storing encrypted data at a node, is thus separated from access to content, i.e. access to the encryption key needed to decrypt that data. Hence, Pesto nodes can exchange storage resources independently from actual content.

The collection of Pesto nodes do not offer its users a single global system view with pre-defined relationships between (subsets of) nodes. This means, for example, that there is no system-wide trusted computing base. In general, no special sets of nodes are defined a priori to be more reliable, safer, more secure or more capable of performing any specific task on behalf of a user. Pesto allows the user to define what other nodes may execute specific tasks on his behalf, based on his personal requirements, beliefs and current needs. The design of Pesto carefully separates the different mechanisms a distributed storage infrastructure must support. Because of this, a user can place responsibility of the different tasks on different machines, possibly governed by a variety of administrative domains. In this way, the user is free to set up relationships he is comfortable with and assign tasks to different parts of the infrastructure.

A Pesto node is owned by a user who is authoritive over its local storage resources. A user may acquire the right to use storage resources at remote nodes and he may delegate rights to use his resources to others. We envision that users establish service contracts with other users in a variety of ways; online storage service providers offering services for a monetary fee, cooperative users exchanging storage for storage and companies offering their employees access to its storage resources. However, how such contracts are established is of no concern to Pesto. Such a service contract could take many different forms, from signed paper contracts to electronic contracts or verbal agreements. The crux of the contract negotiation, however, is that a secret (key) is exchanged. This secret is subsequently used to authorize P_2P requests to use the negotiated storage resources.

II. 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 of it, together with all updates made to that file after that. As such, any version of a file can be retrieved again. An application may choose to store the actual difference between the current version and a new one, or it may store the complete new content as a file update. Actually, Pesto puts no constraints to the content of a file update at all and leaves it applications to define their own update and access semantics.

A file update is immutable and is identified by a *globally unique identifier* (GUID) which is a 128-bit random number. Due to the random selection from an immense name space, a new GUID can be generated locally with no risk of an identical name existing anywhere in the system.

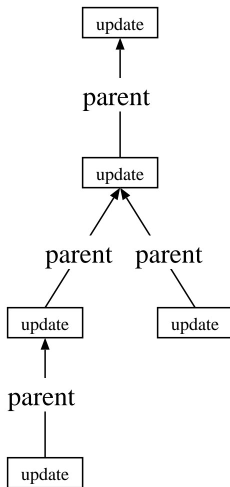

Fig. 1. A file in Pesto is stored as a tree of updates. The arrows are references by GUID to other updates.

Each file update keeps a reference to the previous update to the file, its parent update, such that a file update collection is organized as a file update tree. Or, in other words: A file consists of an ordered set of identifiable updates. Because a file can be replicated, the issue of concurrent updates must be dealt with. However, since each update has an unique name, the two (or more) concurrent updates are identifiable, but they have the same parent (they are derived from the same version). Hence, the ordered set of updates that represents a file is actually a tree, and a file is thus the set of updates, possibly organized as a tree, that together constitutes its content. This is shown in Figure 1.

Because each and every update is uniquely named, any version of a file can be retrieved. The details of how the application chooses to present this fact to the user is not a concern of Pesto. Furthermore, how an application, and its user, chooses to deal with conflicting updates (branches in the version tree), is also left entirely to the application. 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. It does in particular not include consistency control. Pesto thus separates replication from consistency control, and the storage system itself only provides replication.

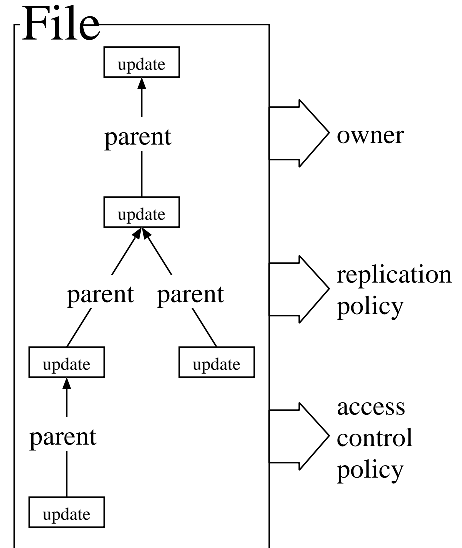

Fig. 2. A file has an owner, a replication policy and an access control policy associated with it. These are references by GUID to other files.

A replication policy and an access control policy can be associated with a file. A replication policy specifies the set of nodes the user expects to store a replica of his file, and it specifies the nodes responsible for the distribution of the replicas to these nodes. An access control policy specifies what credentials a user deems sufficient for other users to be granted access to the content encoded by his file (i.e. who is allowed access to its updates.

We expect the number of different policies to be much smaller than the number of files. An typical user can be expected to define a 'private', 'personal', 'work' and a 'public' access control policy, and maybe a few more. Even though a user's applications may add some generated and/or derived policies of their own, the number of policies a user maintains will be limited. Hence, policies are not stored as part of the files they apply to. Instead, policies are stored in files of their own. A file then references the policies that apply to it by using the GUID of the file which holds the policy. These policy references are specified by the user at file creation time and are immutable A given file can therefore never be associated with a policy with a different GUID. Of course, the content of the policy files themselves can be changed.

A Pesto node maintains a special file that contains a variety of administrative information about it and its owner. This file represents the node and its storage resources and its GUID is used to identify and address the node in the system. As the storage resources of a node are described and represented by this 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.

A file update has a creator associated with it. The creator of a file update is the Pesto node that authorized its creation. The creator of the initial update of a file is said to be the owner of the file. As a creator is a node, it can be identified by its GUID. This GUID 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 with a new GUID.

In short, we can say that Pesto stores the complete update history of files, stores all state of the system in such files. References by GUID are used to associate files with each other (see Figure 2 for an overview of these). Distribution of replicas according user specified policies proceeds whenever communication between source and destination node is possible, i.e. is independent of any synchronization with other replicating nodes. Note also, that there is no mechanism 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.

III. Cryptography

Pesto provides the basic security mechanisms that its users use as building block to define and enforce their security policies. In addition, Pesto allows the user to specify who else is trusted to enforce a security policy he has defined and thus should be granted the ability to do so. Pesto itself is designed around a very simple security policy: all communication and all stored content is regarded confidential and only accessible by the user that created it. In other words, data not properly encrypted is not accepted by the storage system, and the encryption keys are initially only available to the user. 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 then responsible for securely distributing the keys to these nodes, making use as of the secrets established during service contract negotiation as secure channels.

The main goal of file encryption in Pesto is to separate access to storage (of encrypted data) and access to content (via encryption keys). The encryption scheme should facilitate both (and separate) read and update access control, and it should be possible to delegate authority over access control decisions to others. The encryptions should not make it much harder to increase safety of and availability to content by replication.

A. File Encryption

A file stores its complete update history as a tree of file updates. The content of each file update is encrypted with a different encryption key. Such a key encrypting 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 Figure 1 thus uses five member keys and a single file key. 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. Those who know the file key will thus also have access to all member keys and thus have read access to all file updates.

B. Update Authorization

An update to a file is considered authorized if the request is encrypted with a fresh member key, and that key is found encrypted with the file key for that file. A member key is considered fresh for a file update when it has not been used to encrypt an already existing update of the file. Knowledge of the file key therefore gives a user full read and update access to a file.

However, users can obtain authority to update a file, without getting to know the file key. Somebody with access to the file key, for example the file owner, generates a new GUID and a fresh member key in such a case. He subsequently gives the user this GUID and member key together with that same key encrypted with the file key in question.

The user can then use this member key to encrypt his file update and then let Pesto store and distribute it together with the encrypted member key. The users that know the file key are then able to determine that the file update they receive has been properly authorized, by checking the encryptions and freshness of the member key.

C. Revocation

As a consequence of the above, 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 do so, 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 authorization harmless and so revokes it.

D. Granularity of Access

The encryption scheme presented so far achieves that access control can be exercised at fine granularity. Both read and update access is based on access to member keys (respectively to existing and newly created member keys). That is, the unit of access is the file *update*. As described in Section II, there are no constraints on what the content of a file update actually constitutes. It is left to the application to specify file update semantics. This also means that applications can determine the unit of access control, i.e. the granularity of 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. Pesto itself defines somewhat less conventional update semantics for its administrative and policy files, with as main goal enforcement of access control at a fine enough granularity. These semantics are largely outside the scope of this paper, but we want to give a hint of what is possible here using replication policy files as example.

As described earlier, a replication policy file includes the user-specified list of replicating nodes. Pesto actually stores each member of this list as a separate file update to the replication policy file. This way, an access control policy to this replication policy can be enforced that, for example, prevents one replicating node to find out about the location of other replicas.

E. Access Control Policy

We described in section II how each file references an access control policy by the GUID of the file that contains 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 does an access control policy file contain 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 Figure 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.

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. Not only are there far less keys to protect, more importantly 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 access (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.

IV. Trust

The design of Pesto carefully separates the different mechanisms a distributed storage infrastructure must support. Because of this, a user can place responsibility of the different tasks on different machines (nodes), possibly residing in a variety of administrative domains. In this way, the user is free to set up relationships he is comfortable with and assign tasks to different parts of the infrastructure. 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.

A Pesto node can be assigned one or more rôles by any particular user. A node can be assigned the responsibilities for the tasks of storing files, distributing files and/or enforcing access control to files of the user on behalf of the user. The manner in which encryption is applied in Pesto, reduces assignment of such a rôle to a node to the exchange of a secret encryption key. Assignment of responsibility is limited to a user-specified set of files. In other words, different nodes may be assigned responsibilities for 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 respectively:

Trusted Storage Base

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

As described in Section I, a user shares a secret key with each of his storage providers exchanged during service contract establishment. A request to use the negotiated storage resources (i.e. store a file update) is considered authorized if the request is properly encrypted with this shared key, and accounting by the storage provider shows enough negotiated storage are still unused by the user.

If a user wants to act as storage provider for other users, he does so by specifying an access control policy for his storage resources. How this is implemented is outside the scope

of this paper. It is interesting to note, however, that access to storage resources uses the same mechanisms as access to content. Actually, access control of storage resources is reduced to access control of the file that describes them. This assists in keeping the P_2P protocol simple.

Trusted Replicator Base

The nodes of a trusted replicator base (TRB) are trusted to enforce a replication policy on behalf of the user; that is, each member of a TRB is responsible for the *distribution of replicas* to some subset of nodes in a TSB. 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. As with members of the trusted storage base, a user shares a secret key with each of his replicators. Note that, typically, many nodes in the trusted replicator base for a file will also be member of its trusted storage base.

In order to perform their assigned task, a member of the trusted replicator base 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 authorization when an individual node is removed from a TRB.

Trusted Access Base

The members of a trusted access base (TAB) are trusted to enforce an access control policy on behalf of the user. Actually, two separate trusted access bases can be specified for a set of files, the trusted read access base and the trusted (read and) update access base. Members of the trusted read access base are trusted to handout read access and are handed the read access key in order to enable them to do so. Trusted update access base members are trusted to handout both read and update access and are handed the update access key for that.

Pesto itself is not responsible for enforcing these user-specified access control policies, and thus does not prescribe its format and type of contents. In other words, the user and his applications may specify 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 encryption keys required to enforce them to the relevant trusted access bases.

In summary, Pesto keeps different responsibilities separate, so that the user can allocate them to different, but possibly overlapping parts of the infrastructure. The authority needed for such a responsibility is delegated from the user to an encryption key. This reduces the assignment of responsibilities to a key management problem. It is important to note that Pesto's basic key management mechanisms only reduce the number of keys that a user needs to manage explicitly.

Note that other responsibilities could be defined like, for example, consistency control and hence a trusted consistency base for sets of files accordingly. As consistency requirements are highly application dependent, Pesto only provides a basic synchronization mechanism that applications can use to construct their own consistency protocols. Hence, we do not discuss consistency control in this paper, nor do we discuss possible ways to enhance the security of such protocols (see e.g. [1], [2]).

Pesto assumes the user is able to exchange an initial secret (key) over some sort of secure channel with each node it assigns a task to. Whether such an exchange is performed over a physically secure channel or using some form of public key cryptography and infrastructure, is of no concern to Pesto. See also Section VI for a discussion of this issue.

V. Related Work

The Echo file system of Taos relies on a system-wide trusted computing base [3], [4], [5], [6]. The Bayou [7], Coda [8] and Ficus [9] 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 [10] has similar goals and ambitions as Pesto, but relies on public key cryptography for update integrity checking at clients and is therefore less flexible. OceanStore provides end-to-end security, does not trust the infrastructure with unencrypted content and does not rely on a trusted computing base for this. It does not, however, allow the user to specify what part of the infrastructure he trusts, and for what. OceanStore defines a class of servers that are trusted to carry out protocols on behalf of its users and defines the quality of its persistent storage service, where Pesto lets the user specify these.

SPKI provides for an elegant and simple, but yet flexible mechanism to express authorization which meet several of Pesto's design goals [11], [12]. Hence, SPKI is a suitable tool to use in concert with Pesto. Like Pesto, SPKI takes a decentralized approach. But whereas SPKI is "key-centric", Pesto is "user-centric", and 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 [13]. 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.

VI. Discussion

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 repli-

cation mechanisms do not mandate any hierarchical, fixed or static structure on administrative domains. This makes our system inherently suitable for building personal adhoc infrastructures for sharing and cooperation between individuals, but it is certainly not limited to such. Individual administrative domains can be used as building block to construct larger domains using delegation. This requires cooperation from the individual users; users cannot be forced by the system to delegate their authority to others. Enforcement of mandatory policies within the system would require all nodes to be under full control of a single authority. But, even then, users may evade the mandated controls using channels outside the system. Extending the reach of the mandated policy to include such channels as telephone, paper notes and floppy-disks is infeasible in all environments but the most restricted ones (intelligence agencies, the military, criminal organizations and the like).

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. However, cooperation from non-malicious users may be 'bought' 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. It could then setup its communication infrastructure such that only the servers under its control may be reached through it.

A. *Denial of Service*

Our design is well suited for the construction of publication infrastructures similar to "The Eternity Service" [14] and "The XenoService" [15], 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 assist in preventing the former. Resilience to resource exhaustion attacks can be gained by protecting resource use and/or by ensuring more resources are available than an attacker is able to consume. Resource protection is supported by separating access to storage from access to content and its asynchronous protocol. Its 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 the public TAB with merely the exchange of a single cryptographic key, the amount of available resources can be increased dynamically during an attack to counter the resources consumed by the attacker.

B. *Key Management*

Pesto has been designed around our so-called '*Open-End Argument* design guideline [16]. According to it, *the user* should be solely in charge of important matters such as how identity of users are represented and what kind of credentials are needed to authorize actions. Pesto does not support any notion of authentication of "users". By leaving to the user to decide what credentials are considered "good enough", Pesto does not prescribe the structure and organization of the user community. For example, a "web of trust" structure constructed with PGP [17] could be used as well as some kind of certification authority hierarchy. In other words, the user is free to use any existing infrastructure to realize any means of authentication and authorization he might fancy.

We use SPKI base dinfrastructure 'on top of' Pesto, as basis for more advanced authentication, authorization and access control [11], [12], [5], [18]. This infrastructure is structured as a client-server architecture. While its complete implementation consists of about 30.000 lines of Java code, clients have been written that are small and efficient enough to run on a small hand-held device, like a Palm Pilot [18].

Because SPKI is very flexible indeed, sophisticated 'services' such as on-line verification, revocation, and once-only semantics can be offered; see [12] for details. A user would then perform authentication and authorization based on chains of SPKI certificates and use SPKI certificates to construct his access control policy. Of course, typically, the user will delegate enforcement of such a policy to an application that speaks on his behalf and runs on the member nodes of his trusted access bases. Actually, this same application can be used on storage providing nodes to enforce access control to storage resources. Delegation is performed by handing the application the relevant shared keys used by the storage system.

VII. Conclusions

In this paper, we presented Pesto; a flexible distributed storage system simple enough to support resource-poor devices of mobile users. Pesto allows its users to specify what part of the infrastructure is trusted to perform each specific task on behalf of the user. It supports incorporation of the real-world personal and business trust relationships into the system, while not dictating or assuming such relationships in the design itself. Mechanisms to increase safety and availability have been designed together with security mechanisms, providing ease of management and resilience against user error and violation of trust.

A dual-level encryption scheme makes read and update access control possible without relying on public-key cryptography. Public-key cryptography can easily be used for delegation and authorization on top of Pesto's base security mechanisms. 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 neworks that are semi-partitioned. Support for acquisition of authorization before actual use of it provides solid and secure support for disconnected operation.

The separation of access to data and content resulted in that safety and privacy can be addressed separately. The degree of replication can be increased to obtain better avail-

ability 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.

References

[1] Maurice Herlihy and J. D. Tygar, "How to make replicated data secure," in *Proceedings of Advances in Cryptology, CRYPTO '87*. 1988, number 293 in Lecture Notes in Computer Science, pp. 379–391, Springer Verlag.

[2] Adi Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[3] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson, "Authentication in the Taos operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 1, pp. 3–32, Feb. 1994.

[4] Butler Lampson, "Protection," in *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, Princeton University, Mar. 1971, pp. 437–443, Reprinted in ACM Operating Systems Review, 8, 1, January 1974, pp. 18–24.

[5] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber, "Authentication in distributed systems: theory and practice," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 265–310, Nov. 1992.

[6] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart, "The Echo distributed file system," Technical Report 111, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, Sept. 1993.

[7] Doug B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl. Hauser, "Managing Updates in a Weakly Connected Replicated Storage System," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, Dec. 1995, pp. 172–183.

[8] James J. Kistler and Mahadev Satyanarayanan, "Disconnected operation in the Coda file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 3–25, Feb. 1992.

[9] John S. Heidemann, Thomas W. Page, Richard G. Guy, and Gerald J. Popek, "Primarily disconnected operation: Experience with Ficus," in *Proceedings of the Second Workshop on the Management of Replicated Data*, Nov. 1992.

[10] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao, "OceanStore: An Architecture for Global-scale Persistent Storage," in *Proceedings of ACM ASPLOS*. ACM, Nov. 2000.

[11] Carl M. Ellison, "SPKI Requirements," RFC 2692, The Internet Society, Sept. 1999.

[12] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen, "SPKI certificate theory," RFC 2693, The Internet Society, Sept. 1999.

[13] Jon Howell and David Kotz, "End-to-end authorization," in *In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, Oct. 2000, pp. 151–164.

[14] Ross Anderson, "The Eternity service," in *Proceedings of the 1st International Conference on the Theory and Applications of Cryptology*, 1996.

[15] Jeff Yan, Stephen Early, and Ross Anderson, "The XenoService - A Distributed Defeat for Distributed Denial of Service," in *ISW 2000, IEEE Computer Society, Boston, USA*, Oct. 2000.

[16] Tage Stabell-Kulø, Feico Dillema, and Terje Fallmyr, "The open-end argument for private computing," in *Proceedings of the ACM First Symposium on Handheld, Ubiquitous Computing*, Hans-W. Gellersen, Ed. Oct. 1999, number 1707 in Lecture Notes in Computer Science, pp. 124–136, Springer Verlag.

[17] Phillip Zimmermann, *The official PGP user's guide*, The MIT Press, 1995, ISBN 0-262-74017-6.

[18] Per Harald Myrvang, "An infrastructure for authentication, authorization and delegation," Cand.scient. thesis, Department of Computer Science, University of Tromsø, Norway, May 2000.