A Secure Directory Service based on Exclusive Encryption

John R. Douceur, Atul Adya, Josh Benaloh, William J. Bolosky, Gideon Yuval
Microsoft Research
{johndo, adya, benaloh, bolosky, gideony}@microsoft.com

Abstract

We describe the design of a Windows file-system directory service that ensures the persistence, integrity, privacy, syntactic legality, and case-insensitive uniqueness of the names it indexes. Byzantine state replication provides persistence and integrity, and encryption imparts privacy. To enforce Windows' baroque name syntax – including restrictions on allowable characters, on the terminal character, and on several specific names – we develop a cryptographic process, called "exclusive encryption," that inherently excludes syntactically illegal names and that enables the exclusion of case-insensitively duplicate names without access to their plaintext. This process excludes entire names by mapping the set of allowed strings to the set of all strings, excludes certain characters through an amended prefix encoding, excludes terminal characters through varying the prefix coding by character index, and supports case-insensitive comparison of names by extracting and encrypting case information separately. We also address the issues of hiding name-length information and access-authorization information, and we report a newly discovered problem with enforcing case-insensitive uniqueness for Unicode names.

1. Introduction

This paper details the design of a Windows-compatible file-system directory service that provides data privacy, data integrity, and access control even if the servers that manage the directory are untrusted. Our design uses encryption for protecting data privacy and Byzantine replication for protecting data integrity. Although this basic approach is applicable to any type of generic data-storage service, our directory service must additionally maintain the syntactic legality and case-insensitive uniqueness of filenames, even though it cannot have access to the plaintext of these names. We address these requirements by constructing an encryption process that is inherently incapable of encrypting syntactically illegal names. More precisely, the decryption process will always produce a syntactically legal plaintext name, given any arbitrary bit string as an encrypted input, and the encryption process is simply the inverse of this procedure. Furthermore, the encryption provides a one-to-one mapping from de-cased legal names to their encrypted representations, so directory servers can verify the case-insensitive uniqueness of names within each directory.

The context for this work is Farsite [1], a secure, scalable file system that logically functions as a centralized file server but that is physically distributed among a network of untrusted desktop workstations. For storing both file data and directory metadata, Farsite uses replication to provide reliability and integrity despite the failure or compromise of a subset of replica holders, and it uses encryption to provide privacy. Files and directories have very different properties, so different replication and encryption techniques are applicable to each. Since files are large, they cannot afford a high degree of replication, but since they are opaque to the system, they can be protected by conventional encryption and a cryptographic integrity check. By contrast, directory metadata is relatively small, allowing a greater degree of replication, but it needs to be managed by the system. To facilitate this management, Farsite uses Byzantine state replication [9], which preserves the integrity of any arbitrary sequence of operations on the replicated data, as long as strictly fewer than one third of the replica-holders are faulty or compromised [23].

A file system's directory service provides a named index of *files*, organized into a hierarchy of *directories*. Each directory contains a list of *entries*, and each entry includes a locally unique *name* and a reference to a file or another directory. Associated with each directory is list of *readers*, who are authorized to read entry names; a list of *writers*, who are authorized to add or modify entry names; and one *owner*, who is authorized to change other users' access authorization. A secure directory service must provide the following *access-control semantics*:

- Only a reader can read entry names.
- Only a writer can add or modify entry names.
- Only the owner can grant or revoke read/write access.

These access restrictions must apply not only to other users in the system, but also to the directory service itself. A compromised directory server should not be able to read entry names, add or modify entry names, or grant or revoke read/write authorization.

In addition to access control, we need to maintain the following *correctness properties*:

- No correctly functioning client will ever see a syntactically illegal name in a directory.
- No correctly functioning client will ever see two identical names in the same directory.
- No two correctly functioning clients will ever see different views of the same directory.

Windows' particular syntactic restrictions on directory entry names are as follows [28]:

- A name may not be null.
- A name may not contain any control characters (those with Unicode value less than 32).
- A name may not contain any of the following reserved characters: " * / : < > ? \ |
- A name may not end with a space or a period.
- A name may not match any of the following reserved strings (where *n* is any decimal digit): AUX, COM*n*, CON, CONIN$, CONOUT$, LPT*n*, NUL, or PRN.

Furthermore, for purposes of determining whether two names are identical, character case is ignored.

The above properties would be straightforward for a directory service to enforce if it could see the names of the directory entries. It is far more problematic to address the general case in which the servers are not authorized readers of the directories they maintain. Our solution is a cryptographic process we call "exclusive encryption" because it inherently excludes syntactically illegal names and because it enables the exclusion of duplicate names without access to their plaintext. This process employs several techniques:

- name mapping to exclude reserved strings
- a procedure for separating out case information
- encoding to exclude null names, control characters, reserved characters, and disallowed terminal characters
- a technique for modifying any block cipher to make it surjective for arbitrary-length strings

These techniques, which could be beneficial in other applications besides our secure directory service, are detailed in § 3, and they are combined into the full exclusive encryption process in § 4. In § 5 and § 6, we discuss other issues and related work before concluding in § 7. But first, the following section presents the architecture and design of the secure directory service for which exclusive encryption was developed, illustrating how we can satisfy both the access-control semantics and the correctness properties listed above.

2. Secure directory service design

To more clearly highlight the challenges we face in our design, we reframe the above-described access-control and correctness requirements as six problems that need to be solved:

1. preventing an unauthorized user from reading directory entry names
2. preventing an unauthorized user from modifying a directory
3. preventing a server from reading directory entry names
4. preventing a server from making an unauthorized directory modification
5. preventing an authorized writer from incorrectly modifying a directory
6. preventing the authorized owner from incorrectly modifying a directory

A conventional directory service running on trusted servers can readily address problems 1 and 2 by authenticating and mediating all user requests. Problem 3 can be addressed by encrypting directory entry names and not allowing the server access to the encryption key. Problem 4 can be addressed through the technique of Byzantine fault-tolerance [9], a general and powerful mechanism for constructing replicated state machines that can tolerate arbitrary behavior by any subset of strictly fewer than one third of the replicas. Solutions to problems 5 and 6 – which don't revive the first four problems – are the main contributions of the present paper.

Figure 1 illustrates the architecture of our system. Directory servers are organized into Byzantine-fault-tolerant server groups of size *S*, which can tolerate $T = \lfloor (S-1)/3 \rfloor$ misbehaving servers [23]. All servers in a group maintain identical directory-state information. As the group receives requests from clients, the servers within the group collectively assign an operation number to each request using a Byzantine agreement algorithm, which guarantees that if no more than *T* of the servers are faulty, all correct servers will agree on an order in which to process the received requests [9]. When a server is prepared to begin a transactional state update, it informs all other servers in the group of its readiness. Once a server learns of at least 2 *T* other servers that are ready as well, it commits its persistent state update. Each operation performable by a server group must be defined to have a deterministic effect on the replicated directory state. This combination of consistent ordering of requests, two-phase supermajority commit, and deterministic operation ensures that all correct servers within the group maintain consistent copies of the shared state [9].

When each server completes an operation, it sends a reply to the requesting client. By hypothesis, no more than *T* servers in each group are faulty, so a client that receives *T* + 1 matching replies can be confident that the reply content is genuine. Our design employs the highly efficient Castro-Liskov protocol [9] for its Byzantine state replication, which, due to its complexity and extensive description elsewhere [9, 10, 11], we do not describe further herein. The interested reader can find a wealth of information on Byzantine fault-tolerance [8, 12, 17, 23] and replicated state machines [22, 35] in the literature.

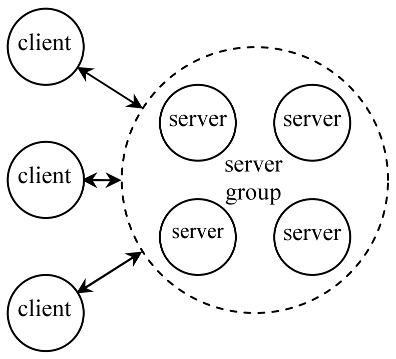

Figure 1. Directory service architecture (*S* = 4, *T* = 1)

For the remainder of this paper, we simply assume that the Byzantine server group acts as a single server that can be trusted to perform the requests it receives. However, we do not assume that it is safe to allow the server group to view or directly modify user-sensitive data, since a single compromised server could inappropriately disclose information.

Each directory has an associated symmetric encryption key that is used for encrypting the directory's name information. This *directory key* is not available (in an unencrypted form) to the servers that maintain the directory metadata. Figure 2 illustrates the state that a server maintains for each directory, which has four components:

- a list of directory entries
- an *access control list* (*ACL*) of *access control entries* (*ACEs*) for authorized readers and writers
- a distinguished access control entry for the owner
- a one-way hash of the directory key

Each directory entry contains the entry name encrypted with the directory key (using the exclusive encryption process described in § 4) and a reference to the file or directory associated with that name. Each access control entry contains a user's public key, a copy of the directory key encrypted with the user's public key, and a bit indicating whether the user has write authorization. The owner is implicitly a writer, so the distinguished access control entry for the owner does not include a write-authorization bit.

Definition 1: An *authorized reader* of a directory is a user for which the directory state includes an ACE that (a) contains the user's public key and (b) contains a ciphertext value that, when decrypted with the user's private key and hashed, yields the directory key hash value stored in the directory state.

Definition 2: An *authorized writer* of a directory is a user for which the directory state includes an ACE that either (a) contains the user's public key and has the write-authorization bit set or (b) contains the user's public key and is the distinguished owner ACE.

Definition 3: The *owner* of a directory is the user for which the distinguished owner ACE in the directory state contains the user's public key.

We illustrate the use of the directory state by detailing the steps involved in a standard set of directory operations: creating a new directory; adding and removing read/write access; reading and listing directory entries; and creating, renaming, and deleting entries.

2.1. Creating a new directory

Olivia, an authorized writer of directory "foo", creates a subdirectory of "foo" with the name "bar" by sending the server group a *create entry* message, which is handled as described in § 2.5. If the creation succeeds, Olivia randomly chooses a new symmetric encryption key for the directory, encrypts the directory key with her own public key, and computes a one-way hash of the directory key. She then sends her public key, the encrypted directory key, and the key hash to the server group, which uses these values to initialize the owner ACE and the directory key hash. At this point, the directory contains no entries, and the ACL contains only the owner ACE.

2.2. Owner operation: add reader/writer

Olivia can make Rita an authorized reader of directory "bar" by encrypting the directory key with Rita's public key and sending Rita's public key and encrypted directory key to the server group, as part of an *add reader* message that she signs with her own private key. (If Olivia has forgotten the directory key, she can retrieve her own ACE from the server and decrypt the directory key using her private key.) The server group verifies the owner's signature and creates a new ACE using the received data.

To make Wallace an authorized writer, Olivia performs a similar procedure but sends the server group an *add writer* message. The server group treats *add reader* and *add writer* messages identically, except for the latter it also sets the write-authorization bit in the user's ACE.

Olivia can make Blaine a blind writer by granting him write authorization but not read authorization. She does this by sending an *add writer* message that contains an incorrect value for the encrypted directory key. Without access to the correct directory key, Blaine is unable to decrypt the entry names; however, the write-authorization bit in his ACL instructs the server group to accept his directory updates. (More on this in § 2.5.)

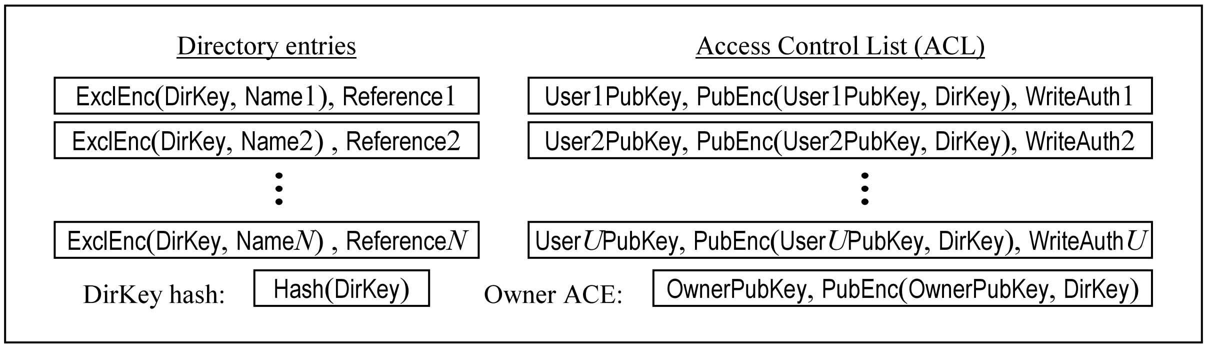
Directory entries
Access Control List (ACL)
ExclEnc(DirKey, Name1), Reference1
ExclEnc(DirKey, Name2) , Reference2
ExclEnc(DirKey, Name*N*) , Reference*N*
User1PubKey, PubEnc(User1PubKey, DirKey), WriteAuth1
User2PubKey, PubEnc(User2PubKey, DirKey), WriteAuth2
User*U*PubKey, PubEnc(User*U*PubKey, DirKey), WriteAuth*U*
DirKey hash:
Hash(DirKey)
Owner ACE:
OwnerPubKey, PubEnc(OwnerPubKey, DirKey)

Figure 2. Directory state maintained by each server

2.3. Owner operation: remove reader/writer

Removing write access is trivially accomplished by clearing the write-authorization bit in the user's ACE.

Removing read access is more involved, because it requires re-keying the directory. To revoke Wallace's read access and leave him as a blind writer, Olivia first retrieves the directory state from the server group. Then, she randomly chooses a new directory key, hashes it, and encrypts it with the public keys of all authorized readers (other than Wallace). She then decrypts and re-encrypts all entry names with the new key and sends all of the new information (except the directory key) back to the server group, which updates its state appropriately.

If Olivia were to revoke Rita's read access, she could – after re-keying the directory and replacing the hash – instruct the server group to remove Rita's ACE, since it provides neither write nor read access. Alternatively, Olivia could leave this ACE in place, even though it has no authorization value (cf. § 5.2 obfuscation techniques).

2.4. Reader operations: read entry / list entries

To read an entry in the directory, Rita first retrieves her ACE and the directory key hash from the server group. She decrypts the directory key using her private key, hashes it, and verifies the hash against the directory key hash from the server group. (If the directory state does not contain an ACE for Rita or if the hashes don't match, then she is – by definition – not an authorized reader.) Rita encrypts the entry name she is looking for – using the exclusive encryption procedure – with the directory key, and sends the encrypted name to the server group, as part of a *read entry* message. If the server group finds a matching encrypted name in the entry list, it returns the associated reference information to Rita.

To list all entries in the directory, Rita begins as above, but rather then sending a *read entry* message containing a specific encrypted name, she sends a *list entries* message to the server group. The group responds by sending Rita a list of all encrypted entry names, which she can decrypt using the directory key.

2.5. Writer operations: create / rename / delete

To create a new entry in the directory, Wallace retrieves his ACE and the directory key hash, decrypts and verifies the directory key, selects a new entry name, encrypts it with the directory key, and sends the encrypted name to the server group, as part of a *create entry* message that he signs with his own private key. The server group verifies Wallace's signature as that of an authorized writer, checks the encrypted name for uniqueness among the list of existing encrypted names, and adds a new entry if the name is unique.

Blaine can also create a new entry in the directory; however, he does not know the name of the entry that he is creating. If he did, then he could use probing to test whether the directory contains a particular entry name, which he should not be allowed to do since he is not an authorized reader. To create a new entry, Blaine generates a random encrypted name and sends it to the server group in a *create entry* message. If the new name is unique, the server accepts it; if it is not, Blaine has to generate a different name, but he learns nothing about the names of entries in the directory, since he does not know what plaintext name his randomly chosen encrypted name would decrypt to.

A rename operation is substantially similar, except that instead of creating a new entry, the server group sets the encrypted name in an existing entry to the new encrypted name.

To delete an entry in the directory, Wallace obtains and decrypts the directory key as above, encrypts the entry name with the directory key, and sends the encrypted name to the server group in a signed *delete entry* message. The server group verifies Wallace's signature and removes the entry with the matching encrypted name, if it exists.

2.6. Directory service security properties

In this subsection, we present and informally justify six security properties maintained by our directory service design. The properties are the access-control semantics and correctness properties itemized in § 1.

In justifying our claims of the following properties, we assume that fewer than a third of the servers in any server group are compromised, so the group can be assumed to provide clients with accurate information and to correctly update the directory state in response to client requests. We assume the security of the underlying cryptosystem, and we assume that authorized users do not deliberately leak information to other users.

Property 1: No one other than an authorized reader can read entry names.

Justification: Entry names are encrypted with the directory key, and are thus unreadable without knowledge of the directory key. In turn, the directory key is stored only in ciphertext form, encrypted with the public key of the authorized readers.

Property 2: No one other than an authorized writer can add or modify entry names.

Justification: The server group updates the entry list only after verifying that the signature on the write request corresponds to a public key in the directory's ACL.

Property 3: No one other than the owner can grant or revoke read/write authorization.

Justification: The server group updates the ACL and the directory key hash only after verifying that the signature on the update request corresponds to the public key in the owner ACE of the directory.

Property 4: No correctly functioning client will ever see a syntactically illegal name in a directory.

Justification: A correctly functioning client will decrypt entry names using the exclusive encryption process, which will produce a syntactically legal plaintext name from any arbitrary ciphertext bit string.

Property 5: No correctly functioning client will ever see two case-insensitively identical names in the same directory.

Justification: The server group ensures uniqueness of the ciphertext entry names, and the exclusive encryption process provides a one-to-one mapping from de-cased legal names to their encrypted representations.

Property 6: No two correctly functioning clients will ever see different views of the same directory.

Justification: The Byzantine protocol guarantees that the server group sends the same state information to all requesting readers. Since a reader is authorized only if the decrypted server key hashes to the server group's directory key hash value, all authorized readers will use the same directory key for name decryption. Thus, all authorized readers will see the same set of entry names.

Properties 4 and 5 rest heavily on the exclusive encryption process, to which we now turn our attention.

3. Techniques for exclusive encryption

This section details a set of techniques that can be used to enforce or enable specific types of exclusions. The general approach is to construct a relation between the domain of syntactically legal names and the codomain of all possible bit strings. This relation must be bijective:

- Injectivity is necessary so that the process is reversible and decryption is possible.
- Surjectivity is necessary for syntax enforcement.
- Injectivity of the inverse is necessary for duplication of plaintext to be detectable by examination of ciphertext.
- Surjectivity of the inverse is necessary for all legal names to be representable.

Although these properties are not all independent of one another, we enumerate them separately to be precise about why we need each one. In particular, surjectivity and inverse injectivity are closely related, but each has a different consequence in our environment.

Exclusive encryption is performed by applying one or more of the techniques described in § 3.1 through § 3.5 (to achieve the desired exclusion), followed by an encryption step (specifically, a block cipher augmented by the technique described in § 3.6). Exclusive decryption is performed by a decryption step (as described in § 3.6) followed by the inverse of one or more of the techniques from § 3.1 through § 3.5.

It is conceptually easiest to understand each of these techniques by appreciating how its inverse (which is performed after decryption) prevents the production of an excluded name. The technique that is applied before encryption is constructed by inverting the inverse.

3.1. Mapping to exclude specific strings

To exclude entire strings (e.g. "AUX") from the set of encryptable names, we construct a bijective mapping from the set of non-excluded strings to the set of all strings. By applying the inverse of this mapping after decryption, any possible decrypted string will de-map to an allowed string. For the result of the inverse mapping to match the original plaintext, the mapping must be applied before encryption.

A simple way to define the mapping is by choosing a (mostly) arbitrary character χ and removing one instance of this character from any string equal to an excluded name followed by one or more instances of χ.

For example, if χ is the underscore character and the name "foo" is excluded, we map "foo_" to "foo", "foo_ _" to "foo_", etc. There is no mapping for "foo", because it is excluded. Non-excluded names are mapped with the identity function, so "bar" maps to "bar".

By construction, any arbitrary string de-maps to an allowed string: Mapped name "foo" de-maps to "foo_". Mapped name "foo_" de-maps to "foo_ _". There is no mapped name that can de-map to "foo". Mapped name "bar" de-maps to "bar".

The choice of character χ is not entirely arbitrary. It must be chosen not to cause one excluded name to map to another. For example, if "fo" and "foo" are both reserved names, the character 'o' cannot be chosen for χ.

3.2. Separating out case information

To enable case-insensitive comparison of names, we decouple the character content of each name from its case information. We do this by creating a string of bits that indicate the case of characters at corresponding positions in the original string. Once we have extracted the case information, we de-case the original string by converting all uppercase characters to their lowercase equivalents. Uppercase characters are thus illegal in the de-cased string, so they are added to the set of excluded characters handled by the technique described in § 3.3–3.5.

When recombining the character and case information, exceptions can be handled in a straightforward manner: For characters that have no case distinction, the case information bit is ignored. If the case information string has fewer bits than the character string has characters, the remainder can be treated as zeroes; and if it has more bits, the excess can be ignored. Case recombination is thus not injective, but this is not a problem since case is irrelevant to duplicate determination.

3.3. Encoding to exclude specific characters

Excluding specific characters (e.g. '/') is more involved than it might seem. One approach is to encode the string using a coding table that includes only legal characters. However, since the count of legal characters is not a power of two, fixed-bit-width encoding is not surjective. If we correct this by multiply encoding some of the characters, we destroy inverse injectivity.

Prefix coding [14] (e.g. Huffman coding) presents a promising avenue, but it is not surjective: It is not possible to determine whether an encrypted string ends with a complete character code. If upon decoding we either discard or arbitrarily complete any partial terminal character, we again destroy inverse injectivity.

To address the last problem, we can truncate the final encoded character in such a way that it can be completed on decode without losing inverse injectivity. In particular, after encoding, we remove all trailing zero bits (if there are any) and the one bit that precedes all trailing zero bits. Before decoding, we append a one bit and as many zero bits as necessary to complete the final character code.

Unfortunately, although this technique preserves inverse injectivity, it loses inverse surjectivity: There is no encoded bit string that corresponds either to the null string or to any string that ends with the character whose code is all zeroes. For our purposes, the former limitation is an advantage, since the null string is not syntactically legal. We address the latter limitation in the following subsection.

3.4. Avoiding the terminal character restriction

The limitation on the terminal character imposed by the above technique would actually be advantageous if Windows' syntax restrictions prohibited only one specific character (such as either space or period) from terminating a name. However, since the number of prohibited terminating characters is not exactly one, this is a problem.

We can remove this limitation by modifying the encoding mechanism. Using the symbol ζ to designate the character whose prefix code is all zeroes, we remove and count all trailing ζ characters from the string to be encoded, encode the remainder of the string as above, and prepend to the encoded string a sequence of one bits equal in number to the count of ζ characters removed from the original string, followed by a zero bit. The encoded string thus begins with a unary representation of the count of ζ characters at the end of the unencoded string.

3.5. Varying exclusions by character position

We can vary the set of allowed characters according to the specific character position, simply by using a different prefix coding table to encode (and decode) the characters in that position. So, for example, we could exclude a certain character from the first position in a string but allow it in all remaining positions.

This technique only works for specific character positions counted from the left of the string, but by reversing the string before encoding it, we can support Windows' restriction on the terminal character of a name.

3.6. Surjective block-cipher encryption

As mentioned above, syntax enforcement requires an encryption method that is surjective. Stream ciphers [27] satisfy this requirement; however, reusing a stream – which would be required for detecting duplicate entry names – leaks a large amount of information and is known to be a severe security weakness.

Conventional block cipher padding techniques [30] are not surjective, but – with one exception – the following technique is: Prepend to the plaintext a one bit preceded by as many zero bits as necessary to bring the total length up to a multiple of the cipher block size. After decryption, discard all leading bits up to and including the first one bit. This technique is surjective except that it cannot produce a padded value whose first block equals zero. This exception is tolerable as long as this case can be identified and rejected by the server.

To enable the exceptional case to be identified, we encrypt the padded string with block cipher F, defined as follows, where E is any standard block cipher encryption:

$$F(k,x) = \begin{cases} 0 & \because \quad x = 0 \\ E(k,0) & \because \quad E(k,x) = 0 \\ E(k,x) & \mathit{otherwise} \end{cases}$$

If the first block of the padded plaintext equals zero, then the first block of the ciphertext equals zero, irrespective of the encryption key.

Decryption is performed with the inverse function F^{-1}, defined as follows, where E^{-1} is the inverse of E:

$$F^{-1}(k,y) = \begin{cases} 0 & \because \quad y = 0 \\ E^{-1}(k,0) & \because \quad E^{-1}(k,y) = 0 \\ E^{-1}(k,y) & \mathit{otherwise} \end{cases}$$

This technique can be applied to cipher modes [30] other than ECB. In particular, it will work correctly with a chaining mode such as CBC as long as the initialization vector is set to zero. The fixed initialization vector can leak information about names with matching prefixes, so if this is a concern, we could apply an all-or-nothing transform [33] to the string before padding. Also, the augmented cipher F is needed only for the first block; subsequent blocks can be encrypted using the unmodified block cipher E, since they are allowed to be zero.

4. Exclusive encryption process

This section specifies the full exclusive encryption and decryption processes used by our secure directory service. The following two subsections present a specific usage of the techniques described in the previous section.

4.1. Encryption

Figure 3 illustrates the full procedure for exclusively encrypting a directory entry name. First, the client maps the name (§ 3.1): If the name equals "AUX", "COM*n*", "CON", "CONIN$", "CONOUT$", "LPT*n*", "NUL", or "PRN", for any digit *n*, followed by one or more underscores, the client removes one trailing underscore; otherwise, it leaves the name alone.

Next, the client separates out case information (§ 3.2): It extracts the case of each character into a separate bit string, and it de-cases the mapped name by converting all uppercase characters to their lowercase equivalents.

Then, it reverses the de-cased name (§ 3.5) so that the following encoding step can restrict the terminal character.

To encode the reversed name, the client first removes all trailing underscores (§ 3.4). Then, it encodes the first character of the reversed name using a prefix coding table that encodes underscore as all zeroes and that does not include codes for '"', '*', '/', ':', '<', '>', '?', '\', '|', uppercase characters, control characters, space, or period (§ 3.5). The remaining characters are encoded using a prefix coding table that is similar, except it includes codes for space and period. From the final encoded character, the client removes all trailing zero bits (if any) and the one bit that precedes all trailing zero bits (§ 3.3). The encoded name is constructed as a one bit for each underscore that was removed (§ 3.4) followed by a zero bit, followed by each encoded character in sequence.

The client pads and encrypts the name using function *F* defined in § 3.6. The case information is also encrypted, but this uses the unmodified block cipher *E*.

After the client sends the encrypted name and the encrypted case information to the servers, the server group verifies the encrypted name by making sure its first block is not equal to zero (§ 3.6). If it is, it rejects the client's request. Otherwise, it performs the requested operation according to the appropriate procedure from § 2.

4.2. Decryption

Figure 4 illustrates the full procedure for exclusively decrypting a directory entry name. First, the client decrypts the encrypted name using function F^{-1} defined in § 3.6 and removes the padding. It also decrypts the case information using unmodified block cipher E^{-1}.

It then appends a one bit followed by a number of zero bits whose count equals the length of the longest prefix code in the coding table (§ 3.3), after which it removes all leading one bits (if any) and the succeeding zero bit (§ 3.4). It then decodes the first character using the coding table that excludes space and period (§ 3.5), and it decodes the remaining characters using the other coding table (§ 3.3). Decoding stops when only zero bits remain. The client then appends an underscore for each leading one bit it removed from the encoded string (§ 3.4).

The client then reverses the decoded name (§ 3.5), and it recombines the case information (§ 3.2).

Finally, the client de-maps the name (§ 3.1) by appending an underscore if the string equals "AUX", "COM*n*", "CON", "CONIN$", "CONOUT$", "LPT*n*", "NUL", or "PRN", for any digit *n*, followed by zero or more underscores.

A client that follows this decryption procedure is guaranteed to see entry names that satisfy the correctness properties itemized in § 1, irrespective of whatever data any other client attempted to send to the directory server group.

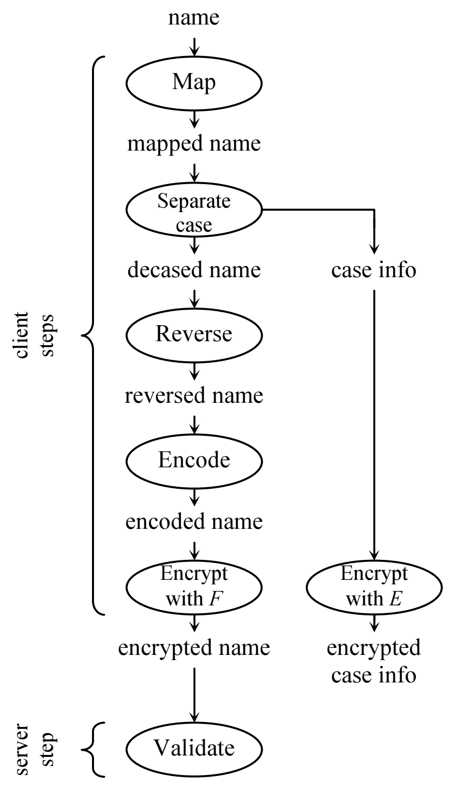

Figure 3. Exclusive encryption procedure

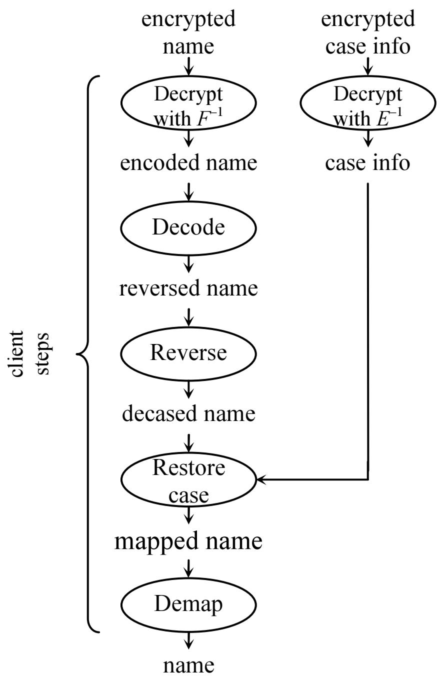

Figure 4. Exclusive decryption procedure

4.3. Examples

Table 1 presents two prefix tables for a very limited alphabet, in which the only legal characters are ‘a’, ‘b’, underscore, period, and space (shown with the symbol). Since period and space are not legal trailing characters, the code table for the first character (after reversing the string) has no codes for these characters.

With neither reserved strings nor case information, and using an identity function as a 4-bit block cipher, Table 2 shows the exclusively encrypted ciphertext for all legal one- and two-character names. It also shows the plaintext for all possible values of a single ciphertext block.

Walking through one example, the name “bb” reverses to itself, and it has no trailing underscores. The first ‘b’ of the reversed name is encoded using the first code table to `1`, and the second ‘b’ is encoded using the standard code table to `01`, from which the trailing one is removed since it is the last character. The encoded name is thus `0` (no trailing underscores) `1` (‘b’) `0` (‘b’ less the trailing one) = `010`. This is encrypted by prepending a one, preceded by no zero bits to pad it up to a multiple of 4 bits. Applying the identity function yields `1010`.

Table 1. Example prefix codes for 5-character alphabet

Character	First prefix code	Std. prefix code
_	00	000
a	01	001
b	1	01
.		10
		11

Table 2. Encryptions and decryptions with identity cipher

Plaintext	Ciphertext
_	0001
a	0100
b	0010
_ _	0011
_a	1100
_b	0110
a_	0010 0000
aa	0010 0100
ab	0001 0100
b_	0001 0000
ba	0001 0010
bb	1010
._	1000
.a	1001
.b	0101
_	0001 0001
a	0001 0011
b	1011

Ciphertext	Plaintext
0000	*illegal*
0001	_
0010	b
0011	_ _
0100	a
0101	.b
0110	_b
0111	_ _ _
1000	._
1001	.a
1010	bb
1011	b
1100	_a
1101	_.b
1110	__b
1111	_ _ _ _

5. Other issues

In this section, we discuss several somewhat tangential issues, such as preventing leakage of name-length information, providing privacy of information other than entry names, dealing with revisions to the Unicode standard, and offering the Windows semantics of making ownership not necessarily imply read or write access.

5.1. Hiding name-length information

Although exclusive encryption prevents unauthorized readers from knowing the name of an entry, it leaks the approximate length of the name. Specifically, the length (in blocks) of the ciphertext name places upper and lower bounds on the length (in characters) of the plaintext name. We can prevent this leakage, at the expense of placing a somewhat quirky restriction on the length of entry names, by modifying the procedure as follows.

First, we must establish a length *L* that all ciphertext names will have. This must be a multiple of the block size, and it in turn limits the length of plaintext names in a convoluted manner: Since characters are encoded using variable-bit-length encoding, the length of the ciphertext is only approximately related to the length of the plaintext.

Before encrypting, rather than padding a name to bring its total length up to a multiple of the cipher block size, we pad it so as to bring its length up to *L*, unless the encoded name is too long, in which case it cannot be encrypted using this technique. The padding is the same as before: a one bit preceded by as many zero bits as necessary.

This technique is surjective except that it cannot produce a padded value that is all zeroes. However, rather than using a modified cipher that enables the server to check for this case, it is simpler to map this special case to a valid legal name that is too long to be encrypted by the standard procedure. One obvious candidate is the string whose encoding is *L* zero bits followed by a one bit.

If we use an all-or-nothing transform [33] to hide partial name matches, it should be applied to the string after it is padded, rather than before; otherwise, it will leak length information through matching zero prefix blocks.

5.2. Obfuscating non-name information

Although our directory service provides privacy against unauthorized readers, this privacy only concerns entry names. It would be nice if we could also prevent leakage of other data, such as file sizes, timestamps, attributes, and directory structure (all of which are above lumped into “reference” information), as well as access authorization. Sizes and timestamps seem impossible to hide from the servers, because the servers themselves directly witness the data represented by these fields, namely when a file is created or written and how much space it consumes. On the other hand, attributes are straightforward to hide using standard encryption. For items that fit neither of these two classes, one approach to improving privacy is obfuscation.

Obfuscating access authorization is straightforward. The owner of a directory can insert ACEs for unauthorized users and set their encrypted directory keys to garbage values, and the server has no way of knowing whether or not the ACEs belong to authorized readers. Furthermore, the owner can insert ACEs for randomly generated public keys that correspond to no actual user, and for these it can even set the write-authorization bit, since no one (other than the owner) knows the corresponding private keys.

Obfuscating structural information about the number of entries in a directory is a considerably harder problem; in fact, we do not currently have a solution. To hide the size of a large directory by splitting it into smaller directories requires a means of partitioning the entries that still enables the servers to enforce directory-wide name uniqueness but does not divulge the logical coherence of the partitions to the servers. Furthermore, maintaining the guarantee that no two correct clients see different views of the same directory requires a means for enforcing consistent access controls among all partitions of a directory, again without betraying this coherence to the servers. Even if we were to devise such a mechanism, it seems likely that traffic analysis could obviate any benefit from this obfuscation.

5.3. Unicode revision and case insensitivity

In developing this directory service, we discovered a problem inherent in the use of case-insensitive comparison for determining duplicate entry names. Since the Unicode standard [41] is evolving, many character codes are not yet defined. Windows allows directory entry names to contain undefined characters (and this is in fact necessary for portability between systems with different language packs installed), but it makes case-insensitive comparisons only for characters that have been defined (and installed). This can lead to a situation in which two names are not at first determined to be identical but then are later judged to be identical following a revision of the Unicode standard (and installation of a language pack).

There are three options for dealing with this issue:

- Abandon the guarantee that no two entries in the same directory have the same name.
- Abandon case-insensitive name comparison.
- Partition the set of allowed characters into two subsets, one case-insensitive and the other case-sensitive.

The first of these options is what Windows does; however, it breaks a reasonable guarantee upon which applications may rely. It is not clear that this poses a true security risk, but since it has been shown that violating a system's assumptions about its input data can provide an entrée for attackers [32], we are uncomfortable taking this approach.

The second option is sensible and internally consistent, but it changes the semantics that most Windows users expect from their file systems, and it threatens backward compatibility for the large installed base of applications that have evolved with the current semantics.

The third option, though somewhat counterintuitive, is our chosen approach. There are two obvious alternatives for selecting a case-insensitive subset: First, we could implement case insensitivity for a particular version of the Unicode standard, (e.g., 3.2.0 [42]), thereby providing broad linguistic coverage. Second, we could select a small subset – such as Basic Latin or Latin-1 – that we expect to cover the majority of use for actual directory entry names. In particular, the Basic Latin subset is so important for backward compatibility that the UTF-8 standard [44] was developed specifically to address this concern. This second alternative has the additional merit of drastically reducing the required size for code tables if a two-stage prefix encoding is employed, since the vast majority of the character space can be encoded with the identity function.

5.4. Ownership without read or write access

Windows allows the owner of a directory not to have read or write access, which our directory service does not. We require the owner to have read access; otherwise, she could not re-key the directory when removing a reader. However, in Windows, an owner always has authorization to grant herself read or write access, so lack of access is merely a convenience to prevent unintentional reading or writing. We can easily provide a similar convenience by restricting an owner's read or write access on the client, since it is irrelevant to true security.

6. Related work

Most distributed file systems, whether server-based [19, 20] or serverless [2, 40], do not address the concern of untrusted remote storage machines, either for privacy or data integrity. Similarly, content-publishing systems [13, 43] and content-indexing systems [18, 31, 34, 39, 45] neither prevent the servers from reading user data nor prevent the publisher from littering the namespace with garbage. Although our design does not prevent a writer from creating nonsensical entry names, it at least restricts the names to a legal syntax and protects their privacy.

The Cryptographic File System (CFS) [6] encrypts both file content and directory entry names on a client machine before writing them to a file server. Each entry name is encrypted using a conventional block cipher (DES [29]) and subsequently encoded in an ASCII representation of its hexadecimal ciphertext value. This encoding technique is not surjective, so syntax enforcement by the server is not possible. However, since CFS does not allow sharing between users, the writer of a directory entry can harm only himself by writing a syntactically illegal name.

Thy Byzantine File System (BFS) [9] replaces an NFS server with a Byzantine-fault-tolerant replica group. Under the assumption that strictly fewer than one third of the servers are compromised (a condition that is provably necessary [23]), BFS guarantees the integrity of file data and directory metadata. However, it does not attempt to provide privacy of file or directory information.

SUNDR [26] is a file system that offers strong integrity and privacy guarantees from the server that provides data storage. It does this by placing full trust in all client machines, which implement the entirety of the file system semantics on top of block-level storage provided by the server. Since the server does not understand the blocks it stores, it cannot guarantee validity of the written data. In addition, since it does not employ Byzantine replication, it is vulnerable to denial-of-service and data-destruction attacks. It does, however, guarantee data consistency by means of all-or-nothing modification semantics.

OceanStore [21] is a distributed object store that uses Byzantine replica groups that understand the semantics of all object updates. It also employs cryptography to protect the privacy of user data, but the design is not yet to a point where it is clear how to harmonize the conflicting goals of privacy and full semantic understanding by the servers.

The Phalanx [25] replication system is an alternative to the BFT toolkit [9] on which our service design is based. Phalanx addresses the issue of "dishonest writers," in the sense of guaranteeing eventual consistency among replicas, but it does not enforce syntactical correctness.

Our directory service provides data privacy through cryptographic means. An alternative approach is to use secret sharing [37] to share the information among the servers in a Byzantine replica group. The Cornell On-line Certification Authority [46] is an example of a system that combines Byzantine fault-tolerance with secret sharing to provide data privacy and integrity, specifically to protect the service's private key. One might imagine a similar approach to addressing directory name integrity, perhaps using verifiable secret sharing [4] to enforce syntax requirements. However, storing different pieces of each name on different servers complicates the design, because Byzantine replicas must be exactly identical.

There has been some other research on performing operations on encrypted data. Song et al. [38] developed techniques for performing searches on encrypted data. Convergent encryption [16] enables identification and coalescing of duplicate files encrypted with different keys. Restrictive blind signatures [7] enable a signer to sign data that it cannot read, while permitting the signer to constrain the structure of the data it signs.

The exclusive encryption process augments a standard block cipher to make it surjective. BEAR and LION [3] and BEAST [24] are block ciphers that have a variable block size and are surjective. Hasty Pudding [36] has a block size that is not only variable but that can even support fractional block sizes. An advantage of our construction is that rather than introducing a new cipher whose security may be in doubt [15], it employs any extant block cipher, some of which have withstood extended cryptanalytic scrutiny [27].

Black and Rogaway [5] present three methods for encrypting an arbitrary finite domain, using constructions based on any extant block cipher. However, without the name-length restrictions introduced (as a side effect) in § 5.1, our encryption domain is infinite.

7. Summary and conclusions

In this paper, we presented the design of a secure, remote, file-system directory service. Our design provides privacy of directory entry names not only from users who are not authorized readers but also from the servers that implement the directory service. In a similar vein, it provides persistence and integrity of directory data despite attempts at destruction or modification either by users who are not authorized writers or by a small fraction of the implementing servers. Furthermore, it enforces syntactic legality [28], uniqueness, and view consistency of directory entry names.

Our service provides privacy through encryption and persistence and integrity through Byzantine fault-tolerance [9]. To enable the enforcement of name syntax and uniqueness without divulging name information to the servers, we developed an encryption procedure – which we call "exclusive encryption" – that is inherently incapable of encrypting syntactically illegal names and that enables a server to check for case-insensitive name uniqueness by examining only ciphertext.

The exclusive encryption process includes several steps, each of which enables a different type of exclusion. To exclude specific strings, it constructs a simple mapping from the set of allowed strings to the set of all strings. To exclude specific characters, it constructs a prefix encoding for all legal characters, amended with a special affix and terminus to maintain the required invariants. The coding can be varied by index to support different restrictions at different character positions. To support case-insensitive comparison, case information is extracted and encrypted separately.

Exclusive encryption requires a block cipher encryption function that is surjective. Although several new ciphers with this property have been proposed [3, 24, 36], we developed a construction that can employ (and derive security from) any extant block cipher. Alternatively, we can directly employ any extant block cipher (without our augmentation) by fixing the size of the name ciphertext. This alternative has the desirable property of preventing the leakage of name length information, but it has the somewhat undesirable side effect of placing a hard-to-characterize restriction on the length of entry names.

During our development, we discovered an intrinsic (and, we believe, previously unreported) problem with enforcing case-insensitive name uniqueness, given that the Unicode character set is not yet fully defined. We suggested several possible work-arounds, but the problem cannot be completely circumvented as long as the character set remains in flux.

Although our service's privacy guarantees apply only to directory entry names, we also considered obfuscation of access authorization information, which is reasonably straightforward, and structural information, which seems *a priori* tractable but for which we have not found a solution that satisfies all of our security and correctness properties. We regard the latter of these as an open problem.

References

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, R. P. Wattenhofer, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment", *5th OSDI*, Dec 2002.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, "Serverless Network File Systems", *15th SOSP*, ACM, Dec 1995, pp. 109-126.

[3] R. Anderson and E. Biham, "Two Practical and Provably Secure Block Ciphers: BEAR and LION", 3rd International Workshop on Fast Software Encryption, 1996, pp. 113-120.

[4] J. Benaloh, "Dense Probabilistic Encryption", *Selected Areas in Cryptography '94*, May 1994, pp. 120-128.

[5] J. Black and P. Rogaway, "Ciphers with Arbitrary Finite Domains", *RSA Data Security Conference, Cryptographer's Track*, LNCS 1872, Springer-Verlag, Feb 2002.

[6] M. Blaze, "A Cryptographic File System for Unix", *Ist Computer and Communications Security*, ACM, Nov 1993.

[7] S. Brands, "Untraceable Off-Line Cash in Wallets with Observers", CRYPTO '93, 1993, pp. 302-318.

[8] R. Canneti and T. Rabin. "Optimal Asynchronous Byzantine Agreement", *Technical Report #92-15*, Computer Science Department, Hebrew University, 1992.

[9] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", *3rd OSDI*, USENIX, Feb 1999, pp. 173-186.

[10] M. Castro and B. Liskov, "Authenticated Byzantine Fault Tolerance Without Public-Key Cryptography", *Technical Memo MIT/LCS/TM-589*, MIT LCS, Jun 1999.

[11] M. Castro and B. Liskov, "A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm", *Technical Memo MIT/LCS/TM-590*, MIT LCS, Jun 1999.

[12] M. Castro and B. Liskov, "Proactive Recovery in a Byzantine-Fault-Tolerant System", *4th OSDI*, USENIX, Oct 2000.

[13] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System", *ICSI Workshop on Design Issues in Anonymity and Unobervability*, Jul 2000.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[15] C. D'Halluin, G. Bijnens, B. Preneel, V. Rijmen, "Equivalent keys of HPC", *Asiacrypt 99*, LNCS 1716, Springer-Verlag, 1999.

[16] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, M. Theimer, "Reclaiming Space from duplicate Files in a Serverless Distributed File System", *ICDCS*, Jul 2002.

[17] J. Garay and Y. Moses, "Fully Polynomial Byzantine Agreement for n 3t Processors in t+1 Rounds", *SIAM Journal of Computing*, 27(1), 1998.

[18] Gnutella. http://gnutelladev.wego.com

[19] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier, "Implementation of the Ficus Replicated File System", *USENIX '90*, Jun 1990, pp. 63-71.

[20] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System," Transactions on Computer Systems, ACM, 1988, pp. 51-81.

[21] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, 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", *9th ASPLOS*, ACM, Nov 2000.

[22] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *CACM*, 21(7), 1978.

[23] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *TPLS* 4(3), ACM, 1982.

[24] S. Lucks, "BEAST: A Fast Block Cipher for Arbitrary Blocksizes", *Communications and Multimedia Security*, IFIP , 1996, pp. 144-153.

[25] D. Malkhi and M. Reiter, "Secure and Scalable Replication in Phalanx", *17th SRDS*, IEEE, Oct 1998, pp. 51-60.

[26] D. Mazières and D. Shasha, "Don't Trust Your File Server", 8th HotOS, May 2001, pp. 113-118.

[27] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.

[28] Microsoft, "File Name Conventions", MSDN, Apr 2002.

[29] National Bureau of Standards, "Data Encryption Standard", FIPS Publication #46, NTIS, Apr 1977.

[30] National Bureau of Standards, "Data Encryption Standard Modes of Operation", FIPS Publication #81, NTIS, Dec 1980.

[31] S. Ratnasamy, P. Francis, M. Handley, and R. Karp, "A Scalable Content-Addressable Network", SIGCOMM 2001, ACM, Aug 2001.

[32] V. Razmov and D. R. Simon, "Practical Automated Filter Generation to Explicitly Enforce Implicit Input Assumptions", *17th ACSAC*, Dec 2001.

[33] R. L. Rivest, "All-Or-Nothing Encryption and The Package Transform", *Fast Software Encryption 1997*, LNCS 1267, Springer-Verlag, 1997, pp. 210-218.

[34] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems", *Middleware 2001*, Nov 2001.

[35] F. Schneider, "Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial", *Computing Surveys*, ACM, 22(4), 1990.

[36] R. Schroeppel, "An overview of the Hasty Pudding Cipher", AES-submission, http://www.cs.arizona.edu/~rcs/hpc, 1998.

[37] A. Shamir, "How to Share a Secret", CACM, 22(11), pp. 612-613, 1979.

[38] D. X. Song, D. Wagner, and A. Perrig, "Practical Techniques for Searches on Encrypted Data", *Symposium on Security and Privacy*, IEEE, 2000, pp. 44-55.

[39] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications", SIGCOMM 2001, ACM, Aug 2001.

[40] C. Thekkath, T. Mann, and E. Lee, "Frangipani: A Scalable Distributed File System", *16th SOSP*, ACM, Dec 1997, pp. 224-237.

[41] The Unicode Consortium, *The Unicode Standard, Version 3.0*, Addison-Wesley, Feb 2000.

[42] The Unicode Consortium, *Unicode Standard Annex #28: Unicode 3.2*, Mar 2002, http://www.unicode.org/unicode/reports/tr28/tr28-3

[43] M. Waldman, A. D. Rubin, and L. F. Cranor, "Publius: A Robust, Tamper-Evident Censorship-Resistant Web Publishing System", 9th USENIX Security Symposium, Aug 2000, pp. 59-72.

[44] F. Yergeau, "UTF-8, a Transformation Format of ISO 10646", *RFC 2279*, Jan 1998.

[45] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing", UCB Tech Report UCB/CSD-01-1141.

[46] L. Zhou, F. B. Schneider, and R. van Renesse, "COCA: A Secure Distributed On-line Certification Authority", *Technical Report 2000-1828*, Department of Computer Science, Cornell University, Dec 2000.