Cinode Documentation

Release 0.1

byo

February 19, 2013

CONTENTS

1 Overview **3**
- 1.1 What the Cinode is? . . . 3
- 1.2 How it's built . . . 3
- 1.3 The security . . . 3
- 1.4 Local access interface . . . 5
- 1.5 Limitations . . . 7
- 1.6 Examples of use . . . 8

2 The blobstore **9**
- 2.1 Blob validation . . . 9
- 2.2 Basic serialization rules . . . 12
- 2.3 Blob types . . . 13
- 2.4 Dynamic Link Blob . . . 15
- 2.5 Summary of blob types . . . 16
- 2.6 Test Vectors . . . 16

3 The Network of Nodes **19**
- 3.1 Node identification . . . 19
- 3.2 Cross-node connectivity . . . 19
- 3.3 Blob requests . . . 19

4 Distributed Hash Table (DHT) **21**
- 4.1 Bittorrent's DHT as a reference . . . 21
- 4.2 Integration with the nodes network . . . 21
- 4.3 Blob and node address space . . . 21
- 4.4 Set of established healthy connections . . . 21
- 4.5 Node ad blob search algorithms . . . 21

5 Indices and tables **23**

Contents:

CHAPTER

ONE

OVERVIEW

1.1 What the Cinode is?

- It is a new kind of network used to host files and run web-like applications
- It's fully distributed system made of resources cumulatively shared by users of the network
- The security and authorization is enforced by best known encryption algorithms
- There's no central management nor any authority
- Initially build on top of the Internet, it can also be used in other communication layers

1.2 How it's built

The smallest building block of the Cinode is a Node. The Node itself is fully functional and allow the user to use it's Local Access Interface. It is using a permanent storage such as a local hard drive to store information. All data is organized in form of data blobs.

Nodes can communicate between themselves using secure connections. Those communication channels are needed to exchange blobs between Nodes. This data migration is used to create backups of data and to create local blob cache that can be quickly accessed by the user.

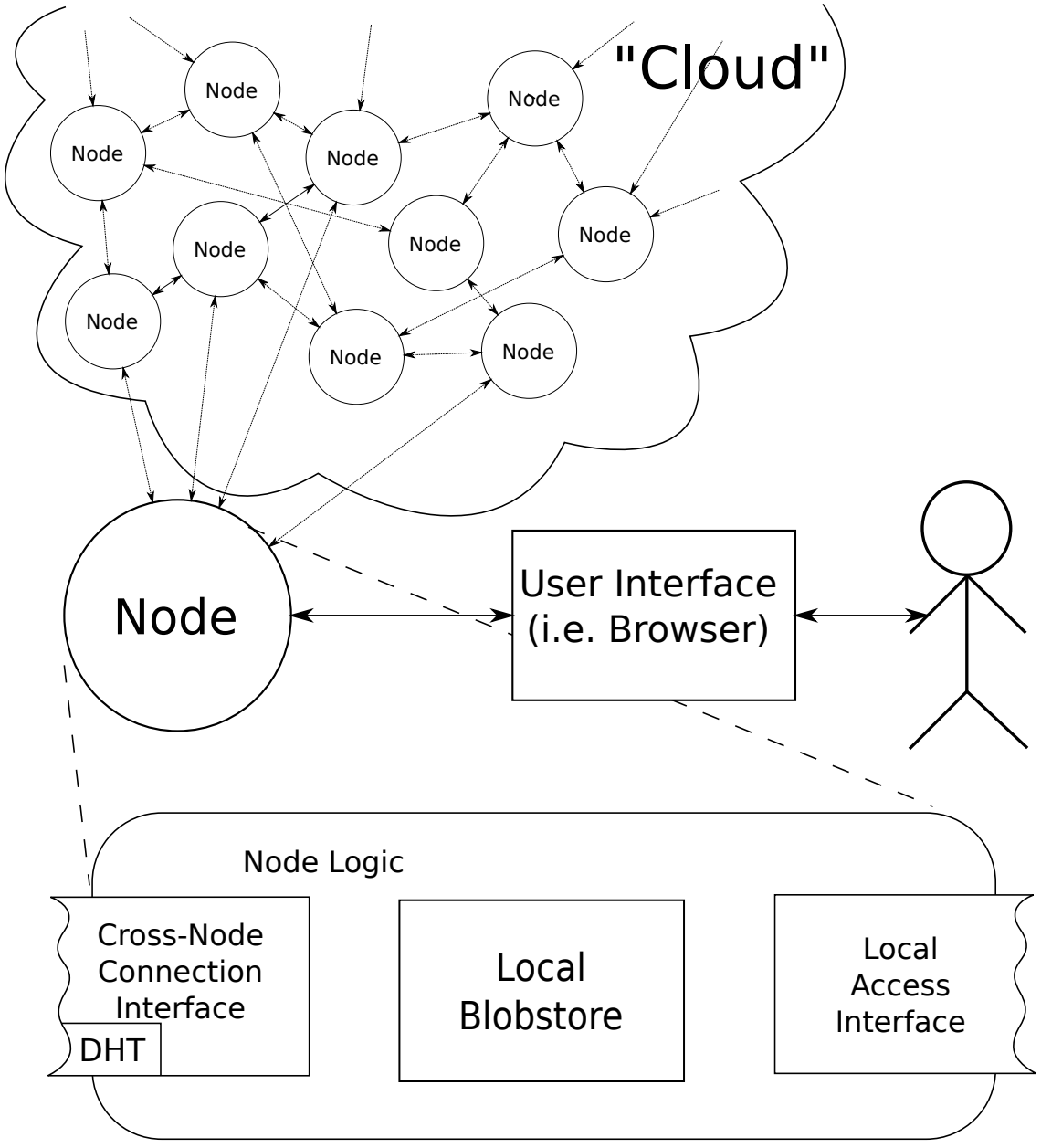
Every data blob is uniquely named using hash-based identifier. This allows a global Distributed Hash Table (DHT) to be used in order to allow searching for a node holding particular blob.

The Local interface available to the user is presented in a form of a filesystem accessible through node's built-in HTTP web proxy. Applications hosted inside Cinode are also loaded from this filesystem. They are executed locally on the user's machine and have access to a part of the filesystem being app's execution environment.

In order to allow users to communicate through Cinode applications, given filesystem directory or file can be shared with another user. Such shared resource can be then plugged into the filesystem of another user becoming available for his own instance of the application. Internally this share is realized by cresting cryptographic connections betwen blobs resulting in moving blobs between nodes. This process is fully automatic from the user's point of view and doesn't need any extra activity.

1.3 The security

Cinode was built with security in mind. The lack of server side forces us to use alternative methods for user authorization. The approach that has been used is based on cryptography.


"Cloud"
Node
Node
Node
Node
Node
Node
Node
Node
Node
Node
Node
Node
Node
User Interface
(i.e. Browser)
Node Logic
Cross-Node
Connection
Interface
DHT
Local
Blobstore
Local
Access
Interface

Every blob in the Cinode network is stored and transferred to other nodes in an encrypted form. The ability to decrypt this data (knowing the decryption key) equals a read access.

In addition, each blob reveals extra cryptographic properties similar to digital signatures. These properties are used to validate blob and make sure that it's content was created, modified or simply accepted by an authorized user. Simply put, it's equal to write access authorization. The creator/modifier/acceptor of the blob must be in a possession of all information needed to make the blob valid such as a private key used to sign blob's content.

The blob validation process does not require read access rights (the data encryption key). By doing so, nodes can safely exchange blobs between themselves without revealing the actual blob content. This allows less trusted 3rd parties to offer services like secure backups, load balancing and blob proxies without the need to reveal the information used by their clients.

1.3.1 Selected encryption algorithms

- Default hash - SHA-512
- Default symmetric cipher - AES-256 in CFB chaining mode
- Default asymmetric cipher - RSA using key of 4096 bits

1.3.2 Consideration of quantum cryptography

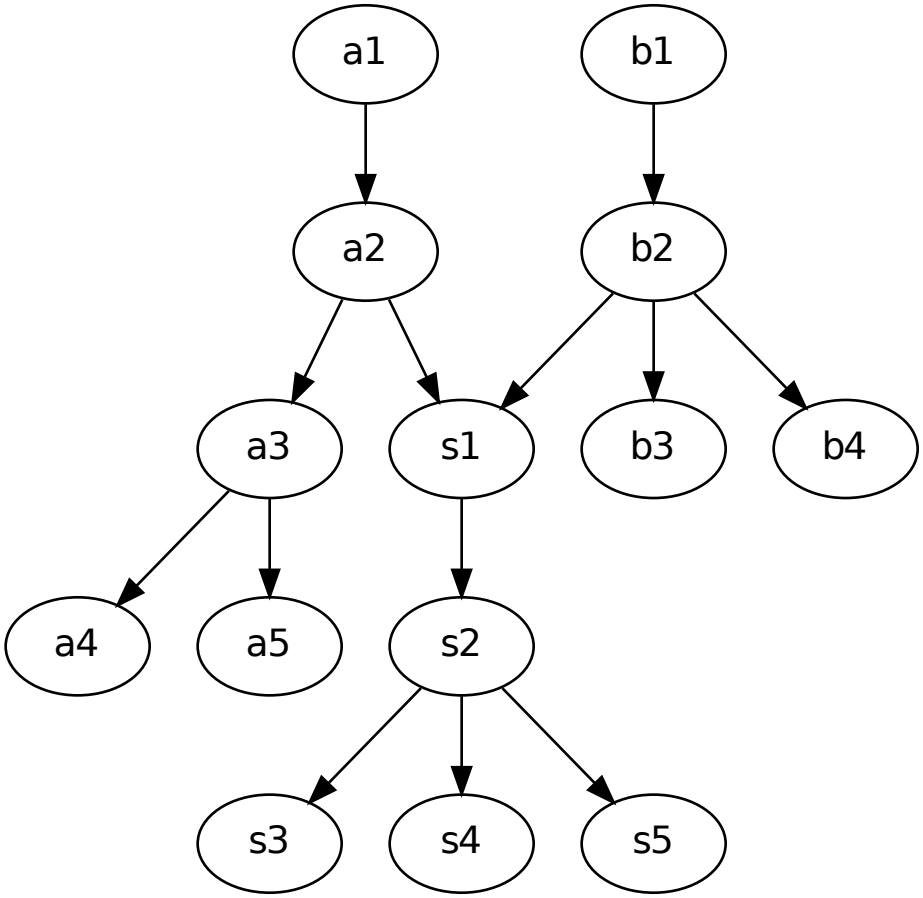
Since algorithms applied in the Cinode may need to stand the test of time, a quantum cryptography must be taken into account. Currently known attacks on symmetric ciphers result in reduction of the algorithm's strengthens by factor of 2 so even if SHA-256 is cracked, it would have the same strength as SHA-128 which is still fairly good. The problem that still persists in the post-quantum era is the ability to break widely used asymetric ciphers such as RSA. For this reason, Cinode should consider using post-quantum cryptography ciphers such as NTRU in order to prepare for the future.

1.4 Local access interface

The user can access the Cinode by a web proxy server built into the node software. It can be disabled creating a node that participates in cross-blob traffic only, a server-like node not accessible by standard user interface.

The HTTP proxy interface provides an easy to use web panel viewable by any modern web browser. It also allows execution of client-side web-based applications written in javascript, native client code or similar technology. Files needed to execute such application are also hosted on the Cinode network removing any dependency for external services.

All information hosted inside the Cinode network is stored inside data blobs. They can form a structure similar to a filesystem due to a presence of directory and link blobs. The actual structure is a bit more complex than a directory tree and is called blob graph, directed cyclic one from the algorithmic point of view. User's Cinode-hosted filesystem is a part of this graph with one blob promoted to be an entry point to user's root directory:



In the example above, a1 and b1 are root blobs for separate users. Each user see his own part of the graph and doesn't see the part to which he has no reference to. For example user of a1-based filesystem doesn't see blobs: b1, b2, b3 and b4. Similarly, the user of b1-based filesystem doesn't see blobs: a1, a2, a3, a4 and a5. The shared part (s1, s2, s3 and s4) can be accessed by both users.

The filesystem can be easily accessed and manipulated through a simple web API provided by the proxy. This is especially useful to Cinode-based applications. Each one of them is executed in a safe sandbox by using a randomized "virtual" hostname. In addition to that, this particular hostname provides an access to a particular sub-directory in the user's filesystem blocking any access outside the sandbox. The randomized hostname for every application and same-origin policy implemented in all modern browsers prevents applications from accessing the information they were not authorized to. It also hardens any sniffing attempts by a malicious software installed locally on user's computer.

Since the user's password usually tends to be the weakest point in nowadays application, Cinode does use cryptographic asymmetric keys instead. In order to login, such key must be provided by the user. Similarly to how openSSH keys are protected, the key may require an extra passphrase before it can be used.

The master key and optional password screen is the first location the web proxy shows to the user. After successful login, the user is presented a master panel where he can view installed applications, browse for files, share nodes etc.

1.5 Limitations

The Cinode network is quite unusual approach to handle cloud services. Some of it's properties make it unsuitable for certain types of applications. There are known limitations of the current Cinode architecture. Some of them can be and will be fixed in future versions, others are tightly bound to the uncommon networking model and thus can not be dealt with without changing root Cinode's design decisions.

1.5.1 Latency and lack of realtime connectivity

The network is based on data blobs floating around between nodes and DHT network used to look for them. This solution, similar to the one used in trackerless torrent networks, introduces significant lag between the time a search is initiated and the time of final blob's arrival. It greatly reduces the ability to exchange information between nodes where the delivery time is critical, especially while dealing with a lot of small data packets. To show an example, a low latency is needed in realtime collaboration software that use video and voice chats.

This limitation can be relaxed in the future by introducing direct connectivity between nodes. Cinode may either be used to establish needed realtime data channels or be just a negotiating medium allowing nodes to establish secure direct connection outside Cinode structure.

1.5.2 The always-remember nature of the network

There's no central management unit in Cinode that could force some blobs to be permanently removed. Any blob that was sent to the network may stay there for unspecified amount of time. This introduces potential security threat due to the fact that some ciphers may be cracked over time.

Future versions of the protocol may include the ability to instruct other nodes to remove a particular blob. But even if this method is implemented, the actual fact of removing data can never be guaranteed in 100%.

1.5.3 Sniffing nodes

One form of malicious software existing in the network will be a sniffing node. A base node's functionality may be extended to store as much information from the network as possible, gathering information for statistical analysis and for possible future cracking attempts.

Although current version does implement read access authorization, it's security relies on the unbreakability of encryption algorithms applied there.

Due to the issue just presented, we highly encourage not to put any confidential, personal or sensitive information into the network during the current phase of it's existence. It's highly probable that some security issues are still present in the software. Instead, we propose to put publicly available information to the network first to gain reasonable amount of information used for testing purposes.

Future Cinode versions will implement ability to restrict read access even more so that it's impossible to even download the encrypted content without proving that read access rights have been granted.

1.5.4 Encryption overhead

Cinode design strongly relies on high quality encryption standards. The tradeoff between encryption efficiency and the security is an always-win for the security. The "always-remember" nature of the network requires us to use the highest know security model since the data we store today may be used in future decryption attacks. The increased key sizes and stronger algorithms may result in increased need for computational power.

1.6 Examples of use

Let's take a look at examples of few applications that can be built on top of the Cinode infrastructure.

1.6.1 Photo gallery

The structure of photo albums can very easily be represented in a directory structure of application's local data folder. Albums are represented by folder's directory structure, pictures in the album being simple files. In addition to this information, each photo may have it's metadata folder containing thumbnails, ratings, comments etc.

Sharing given album with a friend is as simple as sharing one particular album's folder with him. Sharing it with write access rights allows us building complex hierarchical albums structure.

1.6.2 Cloud storage drive

The local filesystem can naturally be represented in a structure of Cinode filesystem. The simplest approach then is to create a virtual drive representing such filesystem and access files directly through such virtual drive.

A more advanced approach would require storing metadata information next to each file and directory. This information (last modification time, checksum, version number etc) can be then used to synchronize local folders with those stored inside the Cinode network.

CHAPTER

TWO

THE BLOBSTORE

The blobstore is a central data point in the Cinode network. Nodes share the data between themselves cumulatively creating a global blob namespace.

The data unit in the blobstore is a called a data blob. It's main purpose is to contain some information, usually in an encrypted form. The knowledge of blob's data decryption key equals read authorization. In addition to the content encryption, the blob reveals extra cryptographic properties used to validate whether an authrized user created the content of the blob. This mechanism is used to implement write authorization.

Every blob is uniquely identified by it's blob Id (BID). It's main purpose is to be able to find desired information in a global blob namespace. In addition to the search ability, BID is used in blob validation process to ensure that given blob can not be simply overwritten by generating blob with the same BID.

The BID does not contain any information related to the blob's content interpretation. This kind of information is stored in unencrypted blob's data. This hardens the analysis of one's blob graph. The type information is stored in common header of unencrypted blob's data. This information is also limited to identify main structures forming user's blob graph and is not intended to contain information such as mime type of files. Every type has a unique number assigned, the list of such types is predefined and should not be altered without a general public discussion. The summary of blob types along with nubers assigned to particular types is presented below.

2.1 Blob validation

Each blob must reveal extra cryptographic properties that allow nodes to quickly accept or reject particular blob when received from the network. This requirement is needed to enforce write access authorization. The validation process will basically prevent blobs generated by unauthorized users from spreading through the network.

The validation process must be done in a cryptographic way so that there are no known practical attacks that could make the validation process useless. The BID identifier is also used during this process to prevent overwriting the blob by generating a new one with the same BID.

Next sections describe currently used validation methods. Future versions of the protocol may extend this set by adding validation rules needed by particular blob types.

2.1.1 Hash-based validation (ID: 0x01)

Hash-based validation is the simplest one and is generally applied to all blob types holding content rather than links to other blobs.

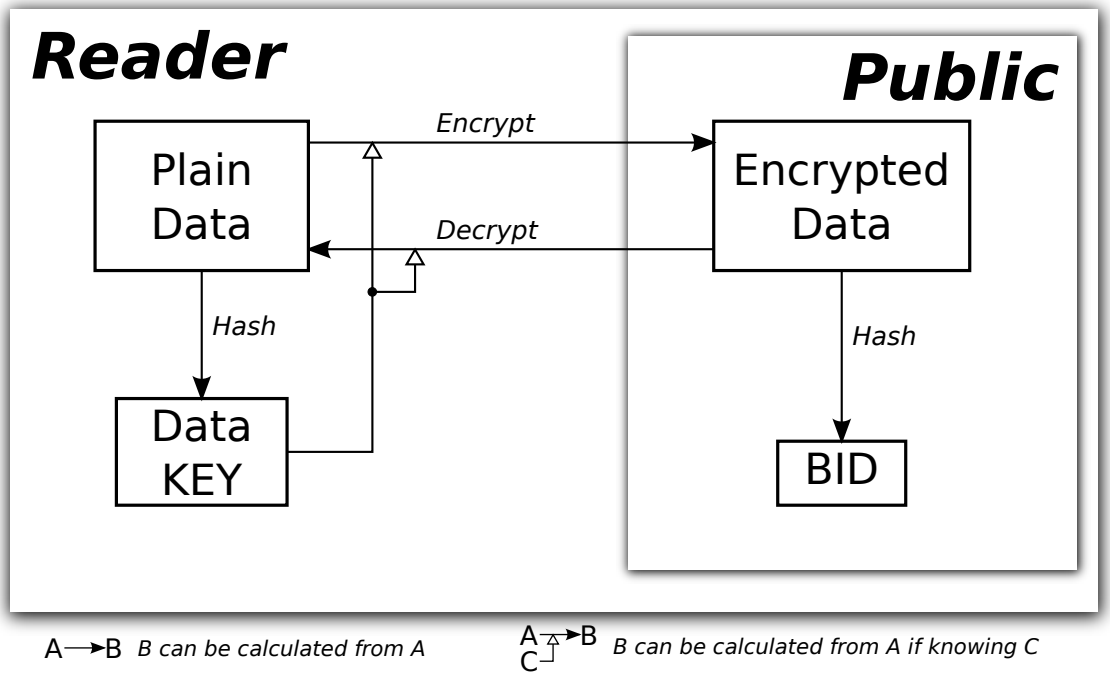
The idea is to use the hash of encrypted blob data as a blob id. The hash used is SHA-512. Using this validation method means that no one can overwrite the content of a blob due to the irreversible nature of cryptographic hashes. This applies to blob's creator too resulting in a write-once like kind of operation.

The process of content change require creating new blob with altered data and replacing all references to the old blob with the new one.

Although this way of handling data comes with many restrictions, it will help dealing with delayed content updates making the network less vulnerable for inconsistencies caused by partial updates (similarly to copy-on-write methods).

The encryption key for the blob's data is generated as the hash of the unencrypted blob's content. This means that given data will always have the same encrypted representation and will allow global anonymous data deduplication.

The IV (initialization vector) for the data encryption cipher is zero in all cases. This doesn't reduce the security since the cipher initialization is based on a hash value that theoretically depends on every bit of the plain data. This key different for every blob is sufficient enough to guarantee semantic security.



The layout of binary exchange form for the blob using this validation method is as follows:

2.1.2 Sign-based validation (ID: 0x02)

This validation method is based on a cryptographic sign of blob's content. Every blob using this method is associated with a public/private key pair. Blob's id is a hash (SHA-512) of the public key (public key FingerPrint) so that the blob with such id can not be overwritten by other users.

Following information is stored for a particular blob:

- Public Key (DER-encoded)
- Digital Sign (calculated from Version Number and Encrypted Data fields)

- Version Number
- Encrypted Data

For validation purposes, all fields apart from the encrypted data are stored in a plain form. The encryption key of a data is based on a hash value (SHA-512) of the Private Key.

The blob can be altered over time resulting in coexistence of multiple contents assigned to the same blob id. Nodes must be able to find the most recent one to keep the network up to date. For this particular reason, each blob stores it's Version Number. Selection of the most recent blob version is a part of blob's validation process.

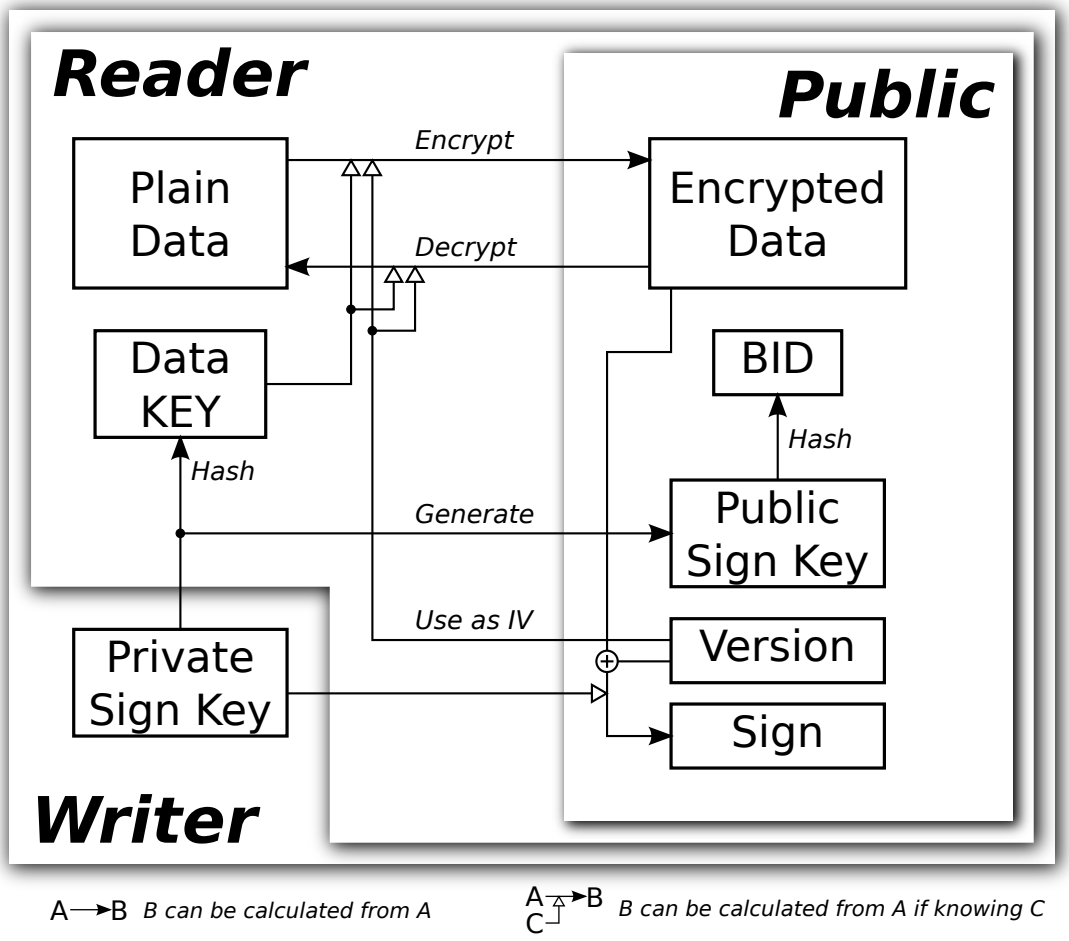
The user authorized to change the blob must know blob's private key and it's most recent Version Number. The new Version Number is generated by adding non-zero positive number to the previous one (it should not be 1 to harden analyzing the number of blob updates). The new Version Number combined with Encrypted Data is signed with the Private Key. New blob is assembled from all needed parts and is propagated to all major cloning nodes (if known).

The validation of a blob is made in the following steps:

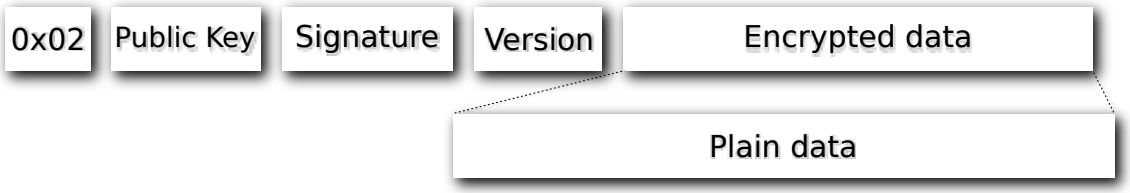
- The blob's Public Key is validated against blob id by comparing key's hash value with the id
- The digital sign of content and version number is validated
- The Version Number is compared with the one of locally stored blob. If Version Numbers are the same, an additional comparison of Digital Sign values is performed in order to deal with same-version attacks.
  - if versions/signs are equal, nothing is done
  - if the version/sign of the received blob is higher than the locally stored one, the local one is replaced by the received one
  - if the version/sign of the received blob is lower than the locally stored one, the local blob is sent back in a reply to blob update request so that the peer can update it's local version

The initialization vector of the cipher is made of the version number by padding it's serialized representation with zeroes or using required number of bytes from the beginning of this representation. Since the Version Number must change every time a new data is introduced, it reveals properties of a cryptographic nounce and thus can be securely used as the IV. This guarantees semantic security.

In case of an attack where the same Version Number is being reused, the attacker must already be in a possession of the private key. This would allow the IV to be reused breaking semantic security. Since the attacker is already in a possession of the private key and thus also the data encryption key, there are no benefits for him other than small disturbance of the validation methods.



The layout of binary exchange form for the blob using this validation method is as follows:



2.2 Basic serialization rules

Blobs use very simple rules for serializing and unserializing information.

2.2.1 Integer

Integers are stored as a sequence of bytes. Each byte consists of two parts. The lower 7 bits are made by cutting next 7 bits of serialized value (starting with lower bits). The eight bit of serialized byte indicates whether the value encoding need more byte(s). The value of 1 means that at least one more byte is needed, the value of 0 means this is the last

byte of serialized data for this integer. The integer must be stored using the shortest sequence of bytes. The value can not be padded with zeroes at the highest bits to increase the length of serialized data.

Here are few examples of data encoding:

Value	Binary value	Encoded form	Binary encoded form
0x00	000000000	0x00	00000000
0x01	000000001	0x01	00000001
0x7F	001111111	0x7F	01111111
0x80	010000000	0x80 0x01	10000000 00000001
0x131	100110001	0xB1 0x02	10110001 00000010

2.2.2 Data buffer

Data buffer consists of two parts:

- integer value indicating the length of buffer in bytes
- sequence of bytes - the content of a buffer

2.2.3 String

String is stored as a data buffer that's an UTF8 representation of the string.

2.2.4 Array

Data starts with an integer being the number of entries followed by entries representation (depends on it's type).

2.2.5 Set

Similarly to the array, data starts with an integer being the number of entries followed by entries representation (depends on it's type).

2.2.6 Map

Data starts with an integer being the number of entries followed by data of each entry. Entry consist of the data of key followed by the data of value.

2.3 Blob types

This section describes all types of blobs available in the Cinode network. Each blob contains information about the interpretation of it's content and validation methods allowed for this blob type.

The information about how the encryption key for blob's data is generated can be found in the documentation of validation methods.

2.3.1 Static File Blob

This blob is used to hold the content of files of size up to 16MB. For files of a larger size, static split file blob must be used.

The content is made of the following fields:

- Type identification: Integer of value 0x01
- The rest of blob's data represents the content of a file, it's not a standard buffer since the size is not preceding the data

Validation method / blob id generator:

- hash-based validation

2.3.2 Static Split File Blob

This blob is used to hold the content of files of size greater than 16MB. The file content is spread among a number of other blobs, each one of Static File type. The size of those blobs must be 16MB with the exception of the last one that's using the remaining part of data.

The content of the file consists of following fields:

- Type identification: Integer of value 0x02
- Cumulative file size: Integer
- Array with a list of blobs containing parts of the file, when concatenated sequentially they form content of the file. Each array entry consists of following fields:
  - BID: String, Id of the sub-blob, the blob must be of type: Static File Blob (0x01)
  - Data KEY: String, Key for sub-blob's data

Validation method / blob id generator:

- hash-based validation

2.3.3 Static Directory Blob

This blob contains information about directory entries assuming that the number of entries is not greater than 1024. In case of blob with greater number of entries, Static Split Directory Blob must be used.

The blob's content consists of following fields:

- Type identification: Integer of value 0x11
- Array of directory entries, sorted by blob names. Each entry consisting of following fields:
  - Name: String (must not be empty)
  - BID: String
  - Data KEY: String

Validation method / blob id generator:

- Hash-based validation

2.3.4 Static Split Directory Blob

This blob can be used to represent directories with a number of entries greater than 1024 by distributing them over multiple separate blobs. The number of sub-blobs - *SubCount* that will be used, is calculated using the following formula:

$$SubCount = \left\lceil \frac{EntriesCount}{1024} \right\rceil$$

In order to distribute entries between blobs, following methods can be used:

- Sorted-Ranges (numeric id of method: 0x01)

  In this method the list of entries sorted by entry name (UTF-8 case sensitive binary sort) is uniformly split into SubCount parts. The assignment of *Nth* entry to a blob of id *BNum* is as follows:

  $$BNum = \left\lfloor \frac{N * SubCount}{EntriesCount} \right\rfloor$$

  N must be in range (0 .. *EntriesCount* -1) inclusively, *BNum* must be in range (0 .. *SubCount* -1) inclusively

  This method should be used when there's a need to scan entry ranges (i.e. list all entries with names between 'alice' and 'bob'). It should be used carefully though since it may reveal some information during sub-blob analysis.

- Hash-Distribution (numeric id of method: 0x02)

  In this method, a hash of the entry name is used to find the desired blob number. The hash is calculated as first four bytes of the SHA-256 function applied on UTF-8 encoded file name. The assignment of blob with name *N* to a sub-blob of id *BMum* is done as follows:

  $$BNum = \left\lfloor \frac{hash(name) * SubCount}{2^{32}} \right\rfloor$$

  This method gives much greater cryptographic security since adding names topologically close to each other usually results in distant sub-blobs.

  This method is preferred and should be used if otherwise not specified.

The content of the blob consists of the following fields:

- Type identification: Integer of value 0x12
- Total number of entries: Integer
- Numeric id of distribution method: Integer of value 0x01 for Sorted-Ranges, 0x02 for Hash-Distribution
- Array with a list of blobs containing entry groups. When all entries from all sub-groups are merged together, they form full list of entries in the directory stored in this blob. Each array entry consists of following fields:
  - BID: String, Id of the sub-blob, it must be of type: Static Directory Blob (0x11)
  - Data KEY: String, Key for sub-blob's data

2.4 Dynamic Link Blob

The purpose of this blob is to link to another blob. It does not use the hash-based validation method so that the value of a particular link blob can change over the time (in a contrast to static blobs). A good example of it's usage is to create a blob id that can be shared with another user ensuring that this user will also see the changes made to this content.

The blob's unencrypted content consists of the following information:

- BID: String, id of the referenced blob
- KEY: String, encryption key of the referenced blob
- RESERVED: Integer of value 0 (reserved for write key propagation)

The referenced blob may also be a link so in order to get to the final blob, one must apply the dereferencing procedure iteratively for all links discovered up to a certain limit of iterations. We suggest a limit of 64 dereferences after which the link may be considered broken.

TODO: Assuming that the link will point to a sub-tree that will have another dynamic links inside, we must be able to pass private keys of those child dynamic links assuming the user has the private key of current link - it's needed in order to cascade write access authorization to such sub-links.

2.5 Summary of blob types

Blob type	TypeId	Validation	Description
Static File Blob	0x01	Hash-based	Store content of a file with size less than 16MB
Static Split File Blob	0x02	Hash-based	Store content of a file with more than 16MB
Static Directory Blob	0x11	Hash-based	Store list of entries inside a directory for number of entries less than 1024
Static Split Directory Blob	0x12	Hash-based	Store list of entries inside a directory for number of entries more than 1024
Dynamic Link Blob	0x31	Sign-based	Store a link to another blob

2.6 Test Vectors

This chapter will contain some test vectors to validate the correctness of implementation. It has been validated using at least two independent implementations. Values are given as hex strings if not specified otherwise.

2.6.1 Empty static simple file blob

Name	Value
File content	
Unencrypted data	`01`
Encrypted data	`eb`
Final blob content	`01 eb`
Encryption Key	`01 7b54b668 36c1fbdd 13d2441d 9e1434dc 62ca677f b68f5fe6 6a464baa decdbd00`
Blob ID	`b4f5a7bb 878c0cec 9cb4bd6a e8bb175a 7ea59c1a 048c5ab7 c119990d 0041cb9c fb67c2aa 9e6fada8 11271977 7b4b80ff ada80205 f8ebe698 1c0ade97 ff3df8e5`

2.6.2 Static file blob with the content "a"

Name	Value
File content	`61`
Unencrypted data	`01 61`
Encrypted data	`8f14`
Final blob content	`01 8f14`
Encryption Key	`01 504ce2f6 de7e3338 9deb73b2 1f765570 ad2b9f2a a8aaec83 28f47b48 bc3e841f`
Blob ID	`c9d30a99 38ecea16 bed58efe 5ad5b998 927a56da 7c8c36c1 ee13292d ec79aa50 c5613fc9 0d80c37a 77a5a422 691d1967 693a1236 892e228a d95ed6fe 4b505d85`

2.6.3 Static file blob with the content "Hello World!"

Name	Value
File content	`48656c6c 6f20576f 726c6421`
Unencrypted data	`01 48656c6c 6f20576f 726c6421`
Encrypted data	`855e296f 95d1eaf3 feb7d48c e0`
Final blob content	`01 855e296f 95d1eaf3 feb7d48c e0`
Encryption Key	`01 ac9d2591 34ccef98 7f9f4df3 115b0b7a 24b379cb ebb2aaa9 1ed811c8 cf5e0907`
Blob ID	`82aeef20 2165cf11 930ea44a 9ad8337a ea355d63 751a7260 552e3e01 4ad6313b ca69c83f a4e35555 31d44a10 25708183 784af0e2 002562b7 260559ce 0e7af262`

2.6.4 Static file blob with the content consisting of all lower-case and upper-case ASCII letters

Name	Value
File content	`61626364 65666768 696a6b6c 6d6e6f70 71727374 75767778 797a4142 43444546 4748494a 4b4c4d4e 4f505152 53545556 5758595a`
Unencrypted data	`01 61626364 65666768 696a6b6c 6d6e6f70 71727374 75767778 797a4142 43444546 4748494a 4b4c4d4e 4f505152 53545556 5758595a`
Encrypted data	`f0ead942 12737b28 60ea35e3 1c7dd176 b5620968 2c3a6792 1d464823 13c245d4 551c765c 3ca851d7 f375911a 66e6b52b 650d51ea c3`
Final blob content	`01 f0ead942 12737b28 60ea35e3 1c7dd176 b5620968 2c3a6792 1d464823 13c245d4 551c765c 3ca851d7 f375911a 66e6b52b 650d51ea c3`
Encryption Key	`01 b11ef5de bd728940 485629e3 42c572bc c5b103d7 b56de27b 07f901b4 abcdb5d4`
Blob ID	`4cfb056a 184d4377 eff9fc3e 8364906a f4b3b3c9 467c2fb8 245382bd d535ea17 f8a63abc 190a9253 9bd92951 52f112d3 365d4910 737b9f9f 3e0eb2f2 eef40648`

CHAPTER

THREE

THE NETWORK OF NODES

3.1 Node identification

TODO

3.2 Cross-node connectivity

TODO

3.3 Blob requests

TODO

CHAPTER

FOUR

DISTRIBUTED HASH TABLE (DHT)

4.1 Bittorrent's DHT as a reference

TODO

4.2 Integration with the nodes network

TODO

4.3 Blob and node address space

TODO

4.4 Set of established healthy connections

TODO

4.5 Node ad blob search algorithms

TODO

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *search*