Hochschule
Augsburg University of
Applied Sciences

Bachelorarbeit

Fakultät für
Informatik

Studienrichtung
Informatik

Florian Schmidt

GNUnet: Analysis of security mechanisms and vulnerabilities

Prüfer: Prof. Dr. Gordon Rohrmair
Abgabe der Arbeit am: 22.10.2012

Hochschule für angewandte
Wissenschaften Augsburg
University of Applied Sciences

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Fakultät für Informatik
Telefon: +49 821 5586-3450
Fax: +49 821 5586-3499

Verfasser der Diplomarbeit:
Florian Schmidt
Sudetenlandstraße 36
85221 Dachau
Telefon:+49 177 7438911
Florian.Schmidt@hs-augsburg.de

Abstract

This thesis describes the security properties of *GNUnet*, a framework for anonymous distributed and secure networking. The first part of this work focuses on the theoretical background and the actual implementation of the security features used by GNUnet. Particularly authentication, deniability and anonymity are analyzed in greater detail, since new and uncommon approaches are used to provide these features. The emphasis of the second part lies on presumed vulnerabilities of these security concepts and possible attacks against them.

Contents

1 Introduction **4**

2 Security mechanisms **6**

2.1 Peer discovery and transport 6

2.1.1 Peer discovery 7

2.1.2 Transport 8

2.1.3 Friend-To-Friend Mode 10

2.2 Authentication and Confidentiality 10

2.2.1 Peer identity 12

2.2.2 Key exchange 13

AES Session Key 14

SetKeyMessage 15

PingMessage 16

Receiving a SetKeyMessage 17

Receiving a PingMessage 18

PongMessage 18

Receiving a PongMessage 19

2.2.3 Link encryption 19

2.3 Anonymity 20

2.3.1 GNUnet's Anonymity Protocol - GAP 21
Routing 22
Indirection and Forwarding 23
Content migration 26
Degree of anonymity 27
2.4 Economic model 28
2.4.1 Trust in GNUnet 30
2.5 Deniability 32
2.5.1 ECRS 32
DBlock and content hash key 33
IBlock 33
KBlock 34
SBlock 36

3 Vulnerabilities 38
3.1 Transport 38
3.1.1 Denial of Service 38
3.1.2 Man-In-The-Middle attack 39
3.2 Authentication and Confidentiality 40
3.2.1 Rewrite Attack 41
3.2.2 Reflection attack 41
3.3 Anonymity 42
3.3.1 Anonymity sets 43
3.3.2 Probabilistic attacks 45
Disclosure Attack 45
Statistical Disclosure Attack (SDA)/ Intersection attack 47
Shortcut attack 49
3.3.3 Other attacks on anonymity 51

(n-1) attack / flooding attack 52
Timing Attack 52
3.4 Economy 53
3.4.1 Sybil attack 54
3.5 Deniability 55
3.5.1 Censoring queries 55
3.5.2 Censoring content 56
3.5.3 Flooding global keyword space 56

4 Conclusion **58**

A Installation **66**
A.1 Installation using release versions 66
Installation of version 0.9.3 of *GNUnet*-gtk: 67
A.2 Installation using latest svn revision 68

B Configuration **69**

1 Introduction

During the last two decades data exchange over the internet has evolved from a technique only required and used by experts to a common way of communications for everyone. While online data exchange became evermore common and important for economy, society etc. also the interest of particular persons and organizations, like governments, media and businesses was evoked for achieving editorial control over parts or all of the information that is exchanged online. Likewise the requirement for anonymous usage of the net has increased significantly, especially over the last decade, as exchange of data which has been conducted through classical ways before, was more and more replaced by online alternatives. Some quite obvious examples involve the publication of and access to uncensored information in totalitarian regimes, traffic shaping executed by internet service providers when using file sharing applications amongst others.

Quite a lot of applications exist nowadays to serve the purpose of anonymous trusted data exchange, like Thor, Freenet, Darknet etc. just to mention the most prominent. The GNUnet application, which is subject of this thesis, especially its file-sharing service, comprises a sophisticated approach to solve the problem of anonymity in a widely distributed peer-to-peer network.

Since GNUnet is still under heavy development this thesis focuses on release

0.9.3. Please note that the protocols used by GNUnet have changed several times between releases, hence it can not be ensured that results of this thesis are applicable to earlier or future versions of GNUnet.

2 Security mechanisms

This chapter describes the techniques that are used by GNUnet to implement its security features. First it is laid out how a peer can discover other peers, how it establishes authenticated connections with those peers and how the messages exchanged over these connections are kept confident. Furthermore it is elaborated how GNUnet provides anonymity for its peers and how it handles resource allocation. Finally the methods used for achieving deniability are discussed.

2.1 Peer discovery and transport

Like most other peer-to-peer system GNUnet is an overlay network that is built on top of the existing internet. Therefore nodes in the overlay can be considered to be connected by logical or symbolic links. To establish such links involved nodes need to know the identity of its counterpart at application level and the transport protocols it uses for communication. GNUnet uses a special type of message to advertise transport information to other peers and an abstraction layer to encapsulate logic for different transport protocols and hiding it from the application, both concepts are specified in [13] and are described below.

2.1.1 Peer discovery

To propagate the peer id, the transport protocols and the transport addresses GNUnet uses a special kind of message, referred to as HELLOs or peer advertisements. These messages contain the identity of the peer and one or more network addresses for different transport protocols.

Listing 2.1: HELLO message

```
struct GNUNET_HELLO_Message
{
        struct GNUNET_MessageHeader header;
        uint32_t reserved GNUNET_PACKED;
        struct GNUNET_CRYPTO_RsaPublicKeyBinaryEncoded
           publicKey;
}
```

This struct is always followed by the actual network addresses which have the format:

1. transport-name
2. address-length
3. address expiration
4. address

When a peer A receives a HELLO message from another peer B it sends a PING message (and also its own HELLO if it hasn't done it yet) back to B. If B answers this PING message with a valid PONG message the advertised combination of peer id, transport protocols and addresses is verified. B also

sends a PING message to A, which in turn also responds with a PONG message (see section 2.2 for details about PING and PONG messages).

2.1.2 Transport

GNUnet supports various transport protocols, like TCP, UDP, HTTP, SMTP and others. To handle this different protocols GNUnet uses a transport abstraction layer (illustrated in Fig. 2.1). This layer processes incoming and outgoing messages and transforms the transport events to GNUnet function calls. The actual file sharing application then communicates with the transport layer via TCP in a client-server like manor and only ever has to handle peer identities and therefore doesn't have to have any knowledge about the transport.

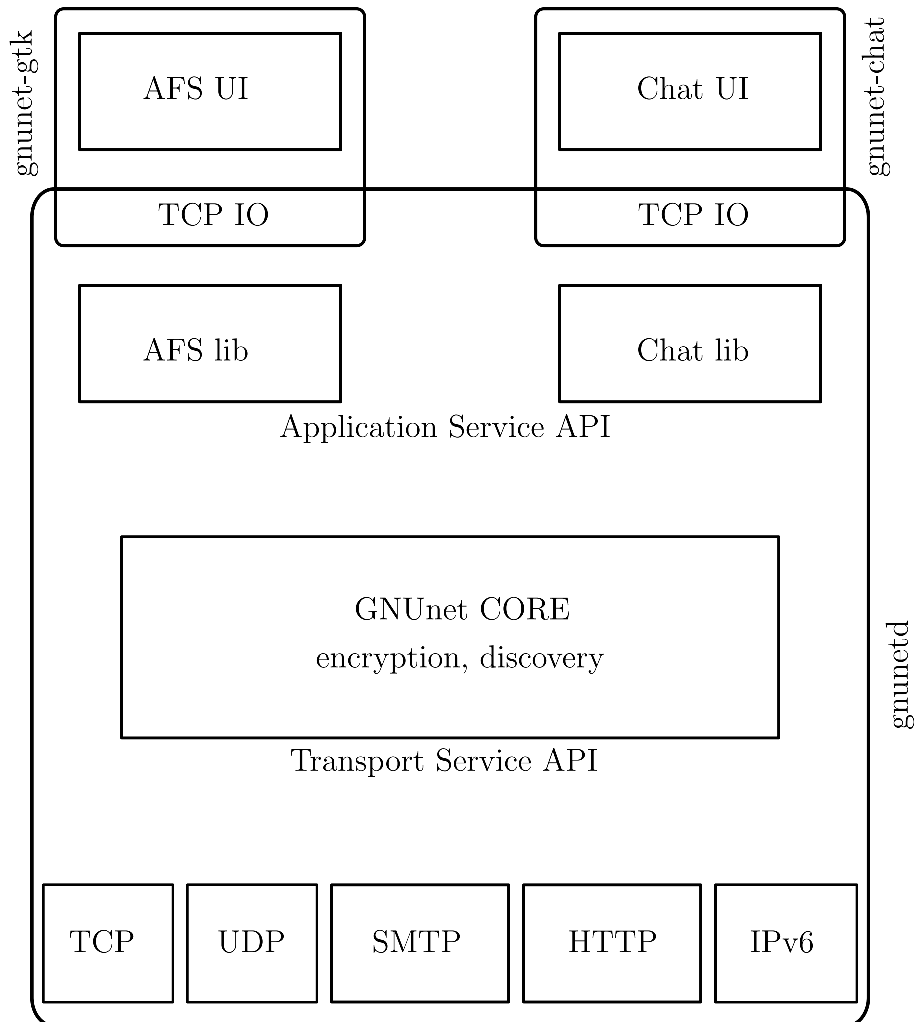


Figure 2.1: GNUnet layers

Peer-to-peer communication is inherently unreliable, because node failures are a common occurrence. Even if those node failures are discarded, a peer might use unreliable transport protocols, like UDP for example. Therefore the transport abstraction layer does not hide network or node failures from the application, to ensure that developers have to consider communications faults in their code.

2.1.3 Friend-To-Friend Mode

GNUnet also provides a so called *friend-to-friend mode*. Contrary to the standard mode, where a peer will connect to any peer, a peer in friend-to-friend mode connects only to peers whose id is contained in a local friend-list file (file containing peer ids in plain text). This white-listing excludes all other peers and increases security for this GNUnet peer, provided the peers in the friends-list are trustworthy. Additionally GNUnet provides a *mixed mode*, which lets a peer connect to arbitrary peers if it has at least a specified number of connections to friends.

2.2 Authentication and Confidentiality

GNUnet uses a duplex three way handshake to establish a session between two peers A and B. During the handshake both peers transmit a message, containing an RSA-encrypted AES session key to their counterpart. The session keys are tested by sending an encrypted message to the other peer which in turn responds with an encrypted message itself. If the reply can be decrypted correctly the reception of the session key is verified and the session is established. After successful authentication the two session keys are used to encrypt all further messages exchanged between the two peers.

RSA is an asymmetric encryption method, which means that separate keys are used for encryption and decryption. These two keys are generated in several steps. First the so called modulus n is calculated by multiplying two large prime numbers. Then Euler's totient function is used to calculate $\Phi(n)$. Eventually the public key exponent e is chosen so that it is coprime to $\Phi(n)$.

Finally the private key exponent d is calculated as the multiplicative inverse of $emod\Phi(n)$. The owner of the key exponent pair A can now transmit his public key $(n, e)_A$ to the intended counterpart B. If B now wants to send a message m to A he can use $(n, e)_A$ to encrypt the message data, which in turn can be decrypted by A using his private key $(n, d)_A$[29].

AES uses the same key for encryption and decryption and therefore represents a symmetric cryptographic technique. Each block of data to encrypt is represented as 4x4 column-major order matrix, which is termed the *state*. Each row of the matrix contains n bytes, where n is the block length divided by 32. The cipher key is generated by using a cryptographically strong random number, which then serves as base to derive a set of keys, called Round keys, with *Rijndael's key schedule*. The first of the derived keys is then combined with each byte of the plain text data using bit-wise xor, this operation is called Round Key addition. Subsequently the *state* matrix is transformed by applying a round function 10, 12, or 14 times (depending on the length of the cipher key), with the final round differing slightly from the previous rounds. First each byte of the *state* is modified independently, by performing a non-linear byte substitution using a substitution table (S-box), this operation is called the *SubBytes* transformation. Secondly the first three rows of the state are shifted cyclically over different numbers of bytes (offsets), this is called the *ShiftRows* transformation. In the next step the data in each column of the *state* is mixed. Each column is treated as a polynomial over Rijndael's finite field $GF(2^8)$ and is then multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$. This is termed the *MixColumns* transformation. Finally the next Round key addition is performed. In the last iteration the execution of MixColumns is omitted. These operations re-

sult in encrypted data which can be decrypted executing the previous steps in reverse order with slight modifications [14].

2.2.1 Peer identity

When a peer is started for the first time, a 2048bit public-private RSA pair is generated, using EME-PKCS1-v1_5 encoding [18]. An SHA-512 Hash of the public key is compiled to serve as the identity of the peer.

Implementation details *GNUnet* uses gcrypt [34] as underlying encryption library. The format for input and output of key and encryption data used by gcrypt are so called S-expressions [28]. *GNUnet* stores an S-expression containing all RSA key data in a data structure [3]. From there the key information is extracted and transformed into a binary format defined by two data structures.

Private key [4] The private key consists of the RSA modulus n with a size of 256 bytes, the decryption key d. Additionally the encryption key e and the prime numbers p, q from which the modulus was calculated are also contained within this data structure.

Public key [5] The public key consist of the RSA modulus n again with a size of 256 bytes and the encryption key e with a size of 2 bytes. The public key part e is always 257, to accelerate the encryption operations.

2.2.2 Key exchange

The peer identity along with bindings for the transport layer is then broadcast to connected peers, this is called peer advertisement and the sent messages are called HELLO messages (see 2.1.1). When a peer receives a HELLO message it sets up two messages. The first message, called SetKeyMessage, contains an AES session key and an RSA signature of the afore-mentioned session key. The second message, called PingMessage contains the encrypted peer identity of the neighbor and a shared secret for later challenge-response authentication. The neighbor, validates the data contained in the PingMessage and replies with a corresponding PongMessage. The PongMessage is again verified and validated by the original peer, by decrypting the shared secret received with the PongMessage. This sequence of operations is depicted in Fig. 2.2.

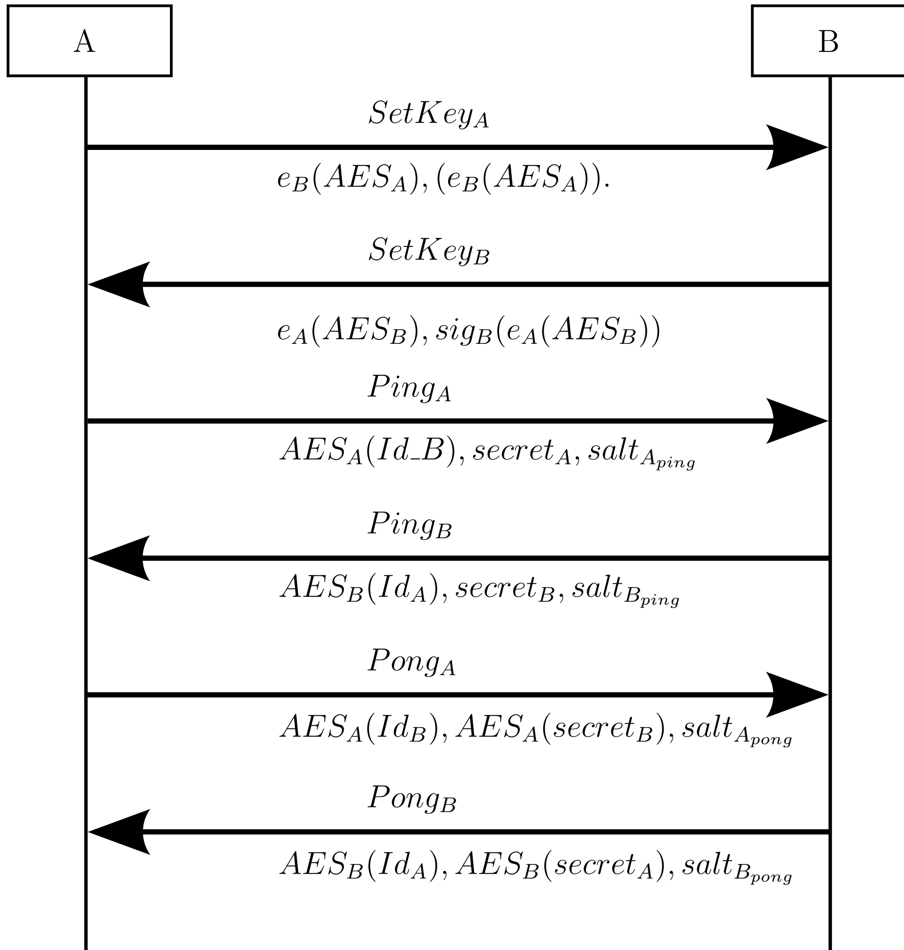


Figure 2.2: Key exchange

AES Session Key

Since asymmetric encryption is costly to calculate, *GNUnet* uses symmetric AES encryption to encrypt data exchanged with other peers. The 256-bit AES session key required for encryption and decryption respectively is generated using a cryptographically strong random function.

SetKeyMessage

The SetKeyMessage is used for transmitting the session key to the other peer. Construction of the message requires two steps, first the session key is encrypted with the public RSA key of the counterpart. Then the encrypted key, along with some additional fields is signed with the private RSA key of the peer. Finally the message is transmitted to the other peer.

Implementation details The field *header*, holds the message header which contains the size and type (GNUNET_MESSAGE_TYPE_CORE_PING) of the message. The field *sender_status* of type *KxStateMachine* gives information about the status of the key exchange. The field *purpose* is of type *GNUNET_CRYPTO_RsaSignaturePurpose*. This struct consists of two fields, the first *size* gives information about how many bits have to be signed. The second one *purpose* contains the actual purpose of the signature (*GNUNET_-SIGNATURE_PURPOSE_SET_KEY* in this case). The field *creation_time* holds the time when the encrypted key was created, which itself is stored in *encrypted_key*. The identity of the recipient is held in the field *target*. The field *signature* holds the RSA signature of the fields *purpose*, *creation_time*, *encrypted_key* and *target*.

Listing 2.2: SetKeyMessage

```
struct SetKeyMessage
{

        struct GNUNET_MessageHeader header;

        int32_t sender_status GNUNET_PACKED;
```

```
        struct GNUNET_CRYPTO_RsaSignaturePurpose purpose;

        struct GNUNET_TIME_AbsoluteNBO creation_time;

        struct GNUNET_CRYPTO_RsaEncryptedData encrypted_key;

        struct GNUNET_PeerIdentity target;

        struct GNUNET_CRYPTO_RsaSignature signature;

};
```

PingMessage

After the SetKeyMessage has been set up, another kind of message is initialized, the PingMessage [6]. The initialization consists of three steps. First a random seed is generated. Then this seed, along with the AES session key for encryption (sk_{dec}) and the identity of the neighbor is used to derive an initialization vector (using HKDF [21]). In the final step the identity of the neighbor is AES encrypted using the initialization vector created in the previous step and sk_{dec}. Then the encrypted identity of the neighbor serves as target of the PingMessage. Also the PingMessage is supplemented with a random value, to serve as shared secret for later challenge-response authentication. Finally the PingMessage, containing the random seed, the encrypted identity and the shared secret is transmitted to the neighbor.

Implementation details The field *header* holds the message header, which contains the size and type (GNUNET_MESSAGE_TYPE_CORE_PING) of the message. The field *iv_seed* contains the salt used for derivation of the initialization vector. The field *target* contains the identity of the recipient and the field *challenge* contains the random value to serve as shared secret.

Listing 2.3: PING message

```
struct PingMessage
{
        struct GNUNET_MessageHeader header;

        uint32_t iv_seed GNUNET_PACKED;

        struct GNUNET_PeerIdentity target;

        uint32_t challenge GNUNET_PACKED;
};
```

Receiving a SetKeyMessage

First the signature sent with the SetKeyMessage is verified using the public key of the neighbor. Then the encrypted session key sent with the SetKeyMessage is decrypted with the private key of the peer to serve as the AES session key for decryption (sk_{dec}).

Receiving a PingMessage

When a peer receives a PingMessage it derives an an initialization vector using sk_{dec}, the seed sent with the PingMessage and the identity of the peer. Then the target field of the Ping Message is decrypted and if the result matches the identity of the peer, reception the PingMessage is deemed successful.

PongMessage

After successful reception of a PingMessage the so called PongMessage is set up. Again a random seed is generated for the PongMessage. Then this seed and the shared secret received with the PingMessage, along with the AES session key for encryption and the identity of the neighbor is used to derive an initialization vector (using HKDF [21]). In the final step the shared secret received with the PingMessage is AES (AES256, Modus: CFB-128) encrypted, using the initialization vector created in the previous step and the session key. Finally the PongMessage is transmitted to the neighbor.

Implementation details The field *header*, holds the message header, which contains the size and type (GNUNET_MESSAGE_TYPE_CORE_PONG) of the message. The field *iv_seed* contains the salt used for derivation of the initialization vector. The field *challenge* contains the encrypted shared secret and the field *target* contains the identity of the recipient.

Listing 2.4: PONG message

```
struct PongMessage
{
```

```
        struct GNUNET_MessageHeader header;

        uint32_t iv_seed GNUNET_PACKED;

        uint32_t challenge GNUNET_PACKED;

        struct GNUNET_BANDWIDTH_Value32NBO reserved;

        struct GNUNET_PeerIdentity target;
};
```

Receiving a PongMessage

When a peer receives a PongMessage it again derives an initialization vector using sk_{dec}, the seed received with the PongMessage, the shared secret sent with the PingMessage and the identity of the peer (= SHA512 hashcode of public key). The encrypted shared secret received with the PongMessage is then decrypted. If the result of the decryption operation matches the shared secret sent with the PingMessage (and the decrypted target matches the peer identity) the key exchange is deemed successful and both peers now have two AES session keys, where $sk_{enc_A} = sk_{dec_t B}$ and $sk_{dec_A} = sk_{enc_B}$.

2.2.3 Link encryption

Only the two peers involved in transmission of a message, sender and recipient, should be aware of the actions being taken. To safeguard the confidentiality between a peer and his neighbor the session keys retrieved at session

initialization are used for link-encryption of all messages exchanged between the two peers.

> *"Step-wise (link-by-link) protection of data that flows between two points in a network, provided by encrypting data separately on each network link, i.e., by encrypting data when it leaves a host or sub-network relay and decrypting when it arrives at the next host or relay. Each link may use a different key or even a different algorithm."* [31]

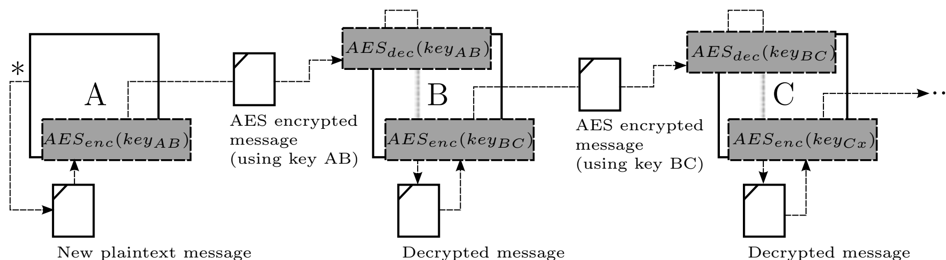


Figure 2.3: Link encryption in GNUnet

2.3 Anonymity

The meaning of anonymity in GNUnet is, that one peer cannot be distinguished from other peers or more exact that the original sender of a message cannot be determined easily or at all. To achieve this goal, a peer in GNUnet uses several mechanisms also common to classical Chaum mixes [9]. These mechanisms encompass the reordering, re-encoding and batching of messages and also the prevention of repeated processing of identical messages.

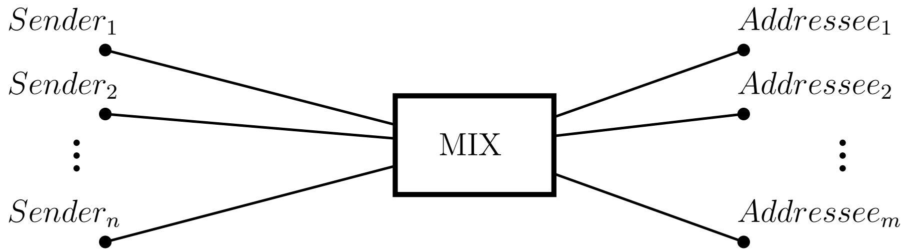


Figure 2.4: Chaum mix

The re-encoding of messages in GNUnet is achieved differently as proposed for classical mixes. Symmetric AES encryption is used for the re-encoding of messages, instead of public key cryptography, to reduce required processing time. Still message contents are signed with the private RSA key of the sending peer. GNUnet also uses some additional methods for anonymization. The original sender information is replaced with the sender information of the forwarding node, if certain conditions are met. Furthermore messages are sent to the next peer with a randomized delay. Content is migrated through the network, thus background noise is generated. All these methods are covered by GNUnet's Anonymity Protocol (GAP) that will be discussed in the following.

2.3.1 GNUnet's Anonymity Protocol - GAP

GAP as presented in [7] was especially designed to match the anonymity requirements of an anonymous peer-to-peer network, especially GNUnet. GAP describes how anonymity is achieved and how peers handle incoming messages. The exchanged messages are for the most part queries for content and replies to those queries, but are supplemented by random noise generated by content migration. A query consists of a resource identifier, the hash code of the content and a node identifier, the hash of the public key of the initiator.

The reply is equivalent to the encrypted content requested in the query. Additionally the GAP protocol uses a pseudo-random value, called time-to-live to prevent loops when routing messages.

Routing

When a peer P receives a query it has to decide to how many and to which of its connected peers $P_1, P_2, \ldots, P_m$ it sends the query to. This decision depends on the available network bandwidth, current CPU load, local credit rating of the sender along with the priority of the query (see 2.4) and a random factor. It is also evaluated, for each connected node, whether its peer id is 'close' to the query hash. To determine the closeness of said hash values a variant of the Pastry algorithm [30] is used. Peers that qualify are preferably chosen as recipients. Nodes that have responded to queries before are more likely to be chosen as one of the addressees as well, this is also referred to as hot path routing.

It is also necessary to prevent queries from looping in the network. Therefore each query is sent with a relative time value, called time-to-live (TTL). It defines how long peers should route replies for this query. When a peer receives a query the TTL is added to the local absolute time and the result is called the local time-to-live (TTL_{loc}). Afterwards the TTL is decreased by a random value. The peer id of the sender, the query hash and the TTL_{loc} are then added to a local routing table. If an entry with the same sender id and query hash already exists, the query is only forwarded if the the original entry is sufficiently old. In this case the TTL_{loc} of the entry is updated in the routing table. TTL_{loc} is also used for ordering received queries, where queries with a higher TTL_{loc} are routed first.

Queries are buffered before they are sent to the selected recipients until the buffer is full or a randomized timer elapses. It has to be noted, that it is possible that no peers are chosen as recipients or that the buffer is discarded, if a peer is considered busy and therefore GAP does not provide reliable delivery of messages.

Replies are padded to uniform size to prevent leaking information about the content of the message. Queries are used for padding, to enhance performance, if possible. Otherwise random noise is generated to pad the message.

Indirection and Forwarding

When node B receives a query q_A from node A it substitutes the node identifier of the query with its own and sends the modified query q_B to node C, as is illustrated in Fig. 2.5.

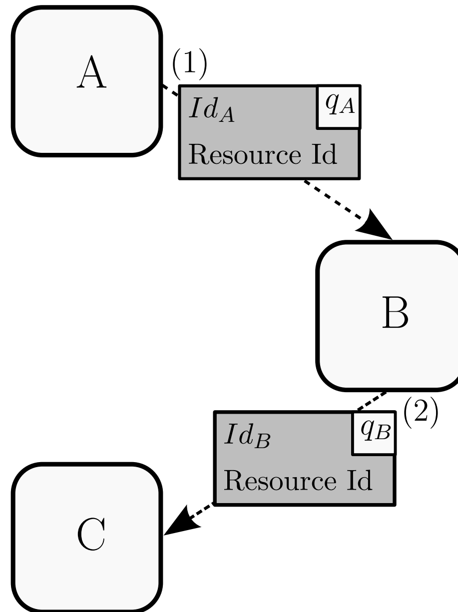


Figure 2.5: Source Rewriting

If Node C holds the requested content it sends reply $r_C(q_B)$ containing the requested content back to node B, which in turn sends reply $r_B(q_A)$ containing the content to node A. This procedure is termed Source Identity Rewriting or indirection and its purpose is to obfuscate the initiator of the query. In the aforementioned scenario C would have no means to determine that A was the originator of the query, since the query only contains the node identifier of B, as is illustrated in Fig. 2.6. On the other hand B knows that the query was originally sent by A, because each node stores queries it has redirected itself in a routing table as mentioned in before (see 2.3.1).

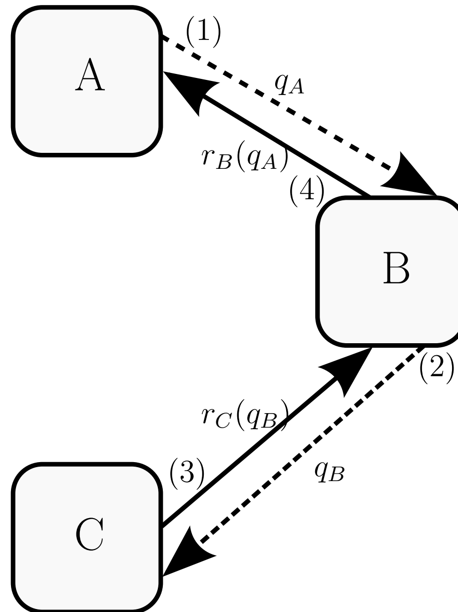


Figure 2.6: Indirection

For further obfuscation of activity, GAP also allows nodes to send queries multiple times, which enables a node to hide queries originating from itself between resent queries, making it harder for an adversary to distinguish between queries originating from a node and queries which are merely resent by the node. To make it even harder for an adversary to determine if queries originate from a node, queries are queued and then sent with a random delay.

Indirection is not mandatory in GAP, which means a node B can choose to leave the node identifier of the query q_A unchanged and just forward it to node C, this is also depicted in Fig. 2.7. In this case the efficiency of the data transfer increases, since the requested content can be sent directly to node A, which might be the requesting node or is at least one hop nearer to the requesting node than B. However since source identity rewriting is

omitted by node B, it possibly damages its own anonymity, because queries forwarded by the node are now distinguishable from queries originating from it.

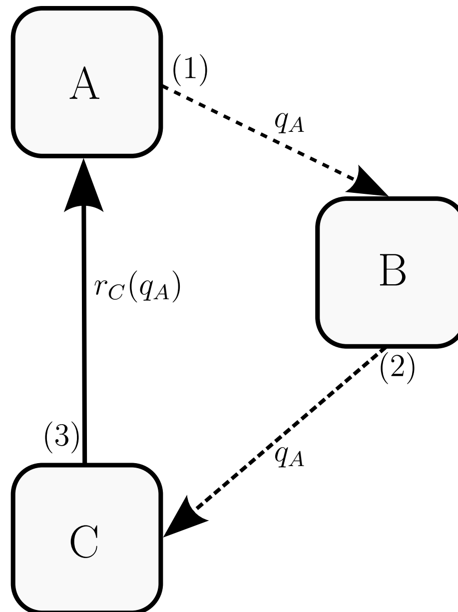


Figure 2.7: Forwarding

Content migration

When a node receives content from another node it has not requested itself, the data is considered valuable and enough local disk space is available it stores the content locally for future requests. This constantly changes the location of the content on the network making it harder for an adversary to pinpoint its exact location. Also if the network is idle nodes will send random content out to adjacent nodes to generate noise, which complicates analysis of activity and also enhances content migration.

Degree of anonymity

To make anonymity measurable it is essential to have a rule for computing the degree of anonymity of a peer P deg_P. deg_P is the inverse of the probability that a message has been initiated by P. deg_P is not a static property and changes depending on the rate of initiated and indirected messages within a specific time interval.

Assuming that peer P indirects the amount of x messages in the given time interval and initiates the amount of y messages itself in the same interval the resulting probability $p_{message_P}$ that an arbitrary messages was initiated by P is $p_{message_P} = \frac{y}{x+y}$ and therefore $deg_P = \frac{1}{p_{message_P}}$. This implies that $p_{message_P}$ increases along with the amount y of indirected messages. On the other hand $p_{message_P}$ decreases along with the amount x of initiated messages. This is also comprehensible intuitively, because $x = 0 \Rightarrow p = 1$ and $y = 0 \Rightarrow p = 0$.

Above example discounts both the fact that the maximal bandwidth b_{max} which is available to P is limited and there's always certain level of background noise in a completely idle GNUnet-network, with the minimal extent of n kpbs. The background noise is caused by content migration, key exchange and forwarding of peer advertisements (HELLOs). If this preconditions are taken into consideration the calculation of deg_P has to be modified slightly.

To simplify the following example it is assumed, that P sends queries with a constant data rate r into the network and doesn't receive any queries in the starting interval i_1 itself. Hence the probability p_{packet_P} that a packet originated from P is $p_{packet_P} = \frac{r}{n}$. If P receives and indirects packets of size t kbps from another peer in the next interval i_2 the probability decreases

to $p_{packet_P} = \frac{r}{n+t}$ and the degree of anonymity increases to $deg_a = \frac{1}{p_{packet_P}}$ accordingly. In some following interval i_k the size of t increases to the extent that the maximum bandwidth capacity of P b_{max} is reached ($b_{max} = r+n+t$). In this state deg_P arrives at the maximum degree of anonymity $deg_{P_{max}}$ that P can achieve with b_{max} and given data rate r.

The assumption that a peer sends queries with a constant data rate r into the network is not met in a realistic scenario, since a GNUnet peer can start the query for a top level IBlock (see 2.5.1) only after it has received the corresponding KBlock or SBlock (also see 2.5.1 and 2.5.1). Similarly a peer can only start the query for an IBlock that is contained deeper within the tree if the parent IBlock has been received previously. The same is also valid for DBlocks (see 2.5.1). Thus the sending and receiving of queries and replies, and therefore r, cannot be constant. The variance of the data rate r is additionally increased, because of buffering and batched sending of messages. It is also possible, that a peer, depending on the bandwidth and CPU load, starts to forward or even drop incoming queries. This decreases the amount of t, which in turn decreases deg_a, assuming the data rate r is constant.

2.4 Economic model

The economic model used by GNUNet as described in [16] differs from other economic models proposed for peer-to-peer systems as laid out in [8], for example, in that it introduces a resource allocation scheme that is not based on classical money-based approaches. Economic models for resource allocations that use money as their currency suffer from a number of shortcomings rendering them ineligible for usage in GNUnet.

First of all money-based models require a central authority that issues the currency used for resource trading. Since GNUnet is a completely decentralized peer-to-peer network, no such entity exists and thus the creation of currency would have to be done by the peers themselves. Indeed there are examples of decentralized currency systems like bitcoins [26], but this approach in turn introduces new problems in the area of anonymity [27]. Secondly the exchange of digital currency, similar to the exchange of physical money, involves the risk of maliciously behaving participants. Let's suppose that peer A advertises its resources for a specific price x. Peer B now wants to use A's resources and agrees to the price. If the money is transferred before B gains access to the resources a could just simply deny B the promised usage or only provide a fraction of the resources. On the other hand, if payment is conducted after usage of A's resources B could just refuse to settle its debt, in part or completely. Because there is no central authority in GNUnet it is hard to impossible to punish such malicious behavior.

Because of above-mentioned limitations of money-based systems GNUnet's economy is based upon *trust*. In this context trust reflects the amount of confidence, deducted from previous interactions with another node, that its resource requests will be satisfied by the opposite node. Contrary to money trust is a localized property in the sense that a node determines itself how trustworthy it deems each connected node to be and in turn cannot control how much it is trusted by these nodes. This also implies that a node has to communicate with another node, before it can assign trust to it or gain its trust. Moreover trust levels can deviate between two nodes, e.g. because the first node has already fulfilled one or more resource requests of the second peer, whereas the second peer has not yet received any requests from the first one. As a result the first node would have a higher trust value at the second

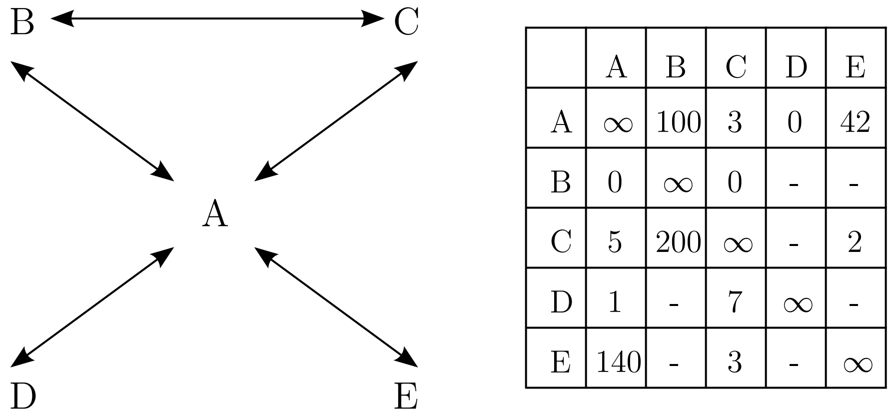
node than the second node has gained with itself.

2.4.1 Trust in GNUnet

When a peer connects to GNUnet for the first time its initial value of trust is zero at all its neighbors. This strategy prevents nodes from trying to gain trust by replacing their peer identity. After having joined the network a GNUnet peer A assesses the contribution of each connected peer $P_1 ... P_n$ with a positive integer value. These values represent the trust within each neighbor. Trust can be gained by participating in the network, more precisely by replying to requests for content or queries in the terms of GNUnet. If a peer delivers content to another peer or, again in GNUnet terminology, sends a reply, its trust value at the recipient increases. The amount of increase depends on the *priority* the query was sent in the first place. The priority is also expressed as a non-negative integer value. Just as the trust value of a peer responding to a query is increased at the side of the originator, the trust value of a peer, having issued a query, is decreased, by the priority value of the said query, on the side of the peer indirecting the query or responding to the query, depending on the load of the responsive peer.

The following simplified abstract example might illustrate this further. Say the trust value $t_A(P_1)$ that peer A applies to P_1 is defined as $trust_A(P_1)$. If A now receives a query with priority $prio_1$ it calculates the effective priority $prio_{Eff} = min(t_A(P_1), prio_1)$ and decreases $t_A(P_1)$ by this value. A can decide to charge P_1 when indirecting the query or when sending the respond. It should be noted that this means that the maximum effective priority $prio_{max}$ that queries P_1 sends to A can have is limited by $t_A(P_1)$. If A itself has sent a query with $prio_2$ to the peer P_2 and receives a reply from it, A increases the

trust value $trust_A(P_2)$ by the value of $prio_2$. In any case A will not disclose how much it charged for a request to discourage manipulation attempts by malicious peers.



	A	B	C	D	E
A	∞	100	3	0	42
B	0	∞	0	-	-
C	5	200	∞	-	2
D	1	-	7	∞	-
E	140	-	3	-	∞

Figure 2.8: Trust in a small GNUnet

It must be pointed out that if a peer is considered idle, which means that its CPU and network load is below a certain percentage p_{busy}, it forwards, or more precisely indirects (see 2.3), queries without considering or reducing the trust value of the sending peer. Still the originator of the request will credit the responder when receiving answer to his request.

If the load of the peer increases beyond p_{busy} however and the peer can only respond to a part of the incoming queries, those with lower effective priority will be dropped and queries with a higher effective priority will be forwarded (or indirected) and charged for as explained above. The reason for the preferred processing of high priority requests is simply that a higher credit rating on the side of the requester is to be gained by the peer when it delivers the response. Since queries with higher priorities are likely to be processed preferably by adjacent nodes, peers will send queries with a sufficiently high priority right from the start to ensure messages are not dropped by their

neighbors.

2.5 Deniability

Even if the anonymity of a peer is broken by an attacker the security architecture of GNUnet provides the feature of credible denial of knowledge about forwarded contents. Deniability is guaranteed because as opposed to intermediaries, the originator of a query, who is also the final recipient of the response, is the only one owning the relevant information to decode the content. Therefore intermediaries cannot be held legally liable for transmitted contents in most jurisdictions. The responding peer can deny knowledge about the content with reasonable credibility, because it could claim to having received the content by migration in the first place. This makes it virtually impossible for the attacker to distinguish responders from intermediaries. A special encoding scheme, as described in [17], has been developed for GNUnet to provide deniability and will be laid out below.

2.5.1 ECRS

Encoding for censorship resistant sharing (ECRS) consists of three components. The first component is the DBlock which corresponds to 32kBit of encrypted data. The second data structure is the IBlock which is similar to UNIX INodes. IBlocks are organized in a tree structure and contain the information necessary to reassemble the data from the distinct DBlocks. The final and topmost component is the KBlock, which contains encrypted meta-data and the encrypted keyword of the query.

DBlock and content hash key

As mentioned above the DBlock corresponds to 32kBit of the original file. The data of the last block is padded with zeros if it is smaller than 32kBit. For encryption a symmetric AES cipher is used. The cipher key of the DBlock containing the encrypted block $E_{K_i}(B_i)$ derived from the plain-text block B_i is $K_i := H(B_i)$, with H being a SHA512 hash function. Also unique identification of an encrypted block $E_{K_i}(B_i)$ is possible by calculating its' query hash $Q_i := H(E_{K_i}(B_i))$. Therefore Q_i can be used to search an encrypted block $E_{K_i}(B_i)$ without revealing K_i. Moreover the key-query pair (K_i, Q_i) can be used to find the encrypted block $E_{K_i}(B_i)$ and decrypt it resulting in B_i. The pair (K_i, Q_i) is called *content hash key* (CHK) [10] and for this reason, the encoding scheme is termed CHK-encoding.

IBlock

Data greater than the block-size of 32kBit has to be encoded by multiple DBlocks. Therefore the same number of CHKs is required to retrieve and decrypt the DBlocks, it is also inevitable to know the order of the CHKs to facilitate reassembly of the original file. For this purpose *ECRS* incorporates the *IBlock*, which contains up to 256 CHKs of corresponding *DBlocks.* This means with one *IBlock* a file of size up to 8 Mbit = 32 Kbit (blocksize) * 256 (max number of CHKs per INode) can be handled. For files greater than this size multiple *IBlocks* are required, one for each 8Mbit block of encrypted data. These *IBlocks* are then CHK encoded similarly to the *DBlocks* and the resulting CHKs are maintained by another layer of *IBlocks.* A file with a size of 32 MBit for example would require 1024 CHKs to be grouped into four *IBlocks.* These four *IBlocks* $I_1 \ldots I_4$ would then be CHK-encoded and the

resulting four CHKs would be arranged into another *IBlock* I_T. Subsequently bigger file sizes result in a larger number of required layers of *IBlocks* which leads to a structure resembling a Merkle-tree [25].

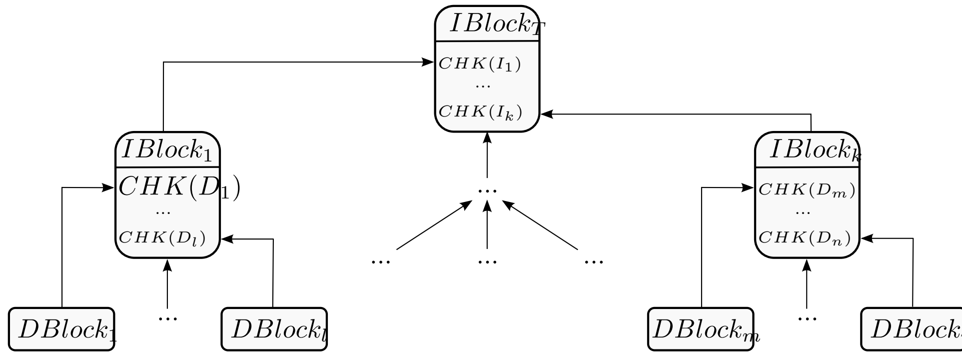


Figure 2.9: IBlocks

KBlock

For usage in a file sharing application it is necessary to be able to search for content with a plain-text keyword. On the other hand keywords should not be exposed to the intermediaries to prevent censorship of specific queries. To provide the functionality of anonymously advertising and searching contents under a keyword a special kind of block is introduced by ECRS, the *KBlock*.

Listing 2.5: KBlock

```
struct KBlock
{
        struct GNUNET_CRYPTO_RsaSignature signature;
        struct GNUNET_CRYPTO_RsaSignaturePurpose purpose;
        struct GNUNET_CRYPTO_RsaPublicKeyBinaryEncoded keyspace
            ;
```

```
        /* 0-terminated URI here */
        /* variable-size Meta-Data follows here */
};
```

A new cryptographic primitive, the *k-deterministic key*, is used for the creation of KBlocks. A *k-deterministic key* is a key pair $(Pub_{H(k)}, Prv_{H(k)})$ derived from a keyword k. Both the public and the private key are generated by using the hash H(k) of the keyword as seed for a pseudo-random number generator. A *Kblock* is then constructed by encrypting the meta-data (MD) required to download the file with a symmetric cipher using H(k) as cipher key. The encrypted meta-data $E_{H(k)}(MD)$ is then signed with the private key $Prv_{H(k)}$. The signed and encrypted meta-data $[E_{H(k)}(MD)]_{Prv_{\mathrm{H(k)}}}$ is then joined with the public key $Pub_{H(k)}$ to form the *KBlock*.

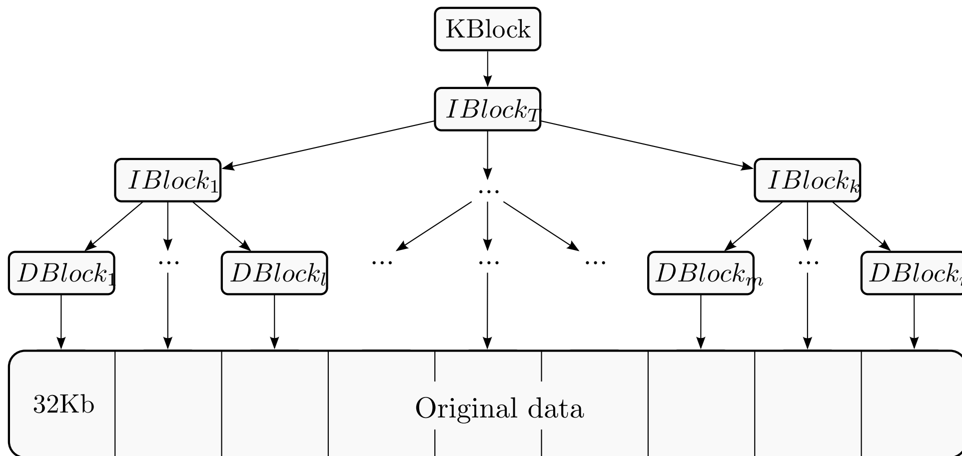


Figure 2.10: Tree of CHK-encoded blocks

SBlock

Any peer in the network can publish content under a certain keyword. Therefore it is to be expected, that searches for certain keywords deliver unsatisfying results, e.g. because content has been published under an inappropriate keyword or it has been published under many different keywords. To alleviate this problem ECRS provides the option to publish and search for contents within a namespace. A namespace consists of a public-private key pair The hash of the public key is used to publicly refer to the namespace. The private key is used to sign content published within the namespace, which allows to verify that the content has been published by the owner.

To incorporate the concept of namespaces into ECRS another kind of block, the *SBlock*, is required. An SBlock is comprised of the encrypted key and query hash (CHK) of the top IBlock together with the encrypted metadata belonging to the content. The hash of the keyword under which the content was published serves as encryption key. Subsequently follows the query identifier of the SBlock, which is the hash of the encryption key.

Listing 2.6: SBlock

```
struct SBlock
{
        struct GNUNET_CRYPTO_RsaSignature signature;
        struct GNUNET_CRYPTO_RsaSignaturePurpose purpose;
        struct GNUNET_HashCode identifier;
        struct GNUNET_CRYPTO_RsaPublicKeyBinaryEncoded subspace
           ;

        /* 0-terminated update-identifier here */
```

```
        /* 0-terminated URI here (except for NBlocks) */
        /* variable-size Meta-Data follows here */
};
```

Since IBlocks and DBlocks are not affected when content is published under a namespace it is possible to publish the same content under several namespaces without the need of additional memory, except for the necessary SBlocks.

To search a namespace only the query identifier along with the hash of the public key of the namespace are required. The query id is used by intermediaries to identify the desired SBlock and the hash of the public key is used to verify the signed contents. Decryption of the contents however is not possible without knowing the original keyword under which the content was published. Therefore only the originator of the query and the original publisher have knowledge about the actual, unencrypted content, thus providing deniability for the intermediaries.

3 Vulnerabilities

In this chapter the security vulnerabilities of *GNUnet* are examined. Therefore various known attacks are tested for applicability against *GNUnet*. Only applicable attacks are described in greater detail to emphasize the boundaries of the inspected features.

3.1 Transport

Theoretically it is possible for an adversary to intercept HELLOs to learn about the used transport system(s) and the related addresses of a node.

3.1.1 Denial of Service

If an attacker knows the transport addresses used by a peer and also controls a (high) number of nodes on the network he can make his nodes drop the traffic of the peer. This wouldn't necessarily lead to a complete denial of service for the peer, if not all his neighbors are controlled by the adversary, but would impair network efficiency and redirect load onto non-malicious neighbors.

Assuming that a peer A has three neighbors, a regular peer B and two malicious peers C_m and D_m, as illustrated in Fig. 3.1. The malicious peers drop all packets from and to A. This means A can only send queries into the network and retrieve data from the network via B and thus bandwidth is reduced and delay is increased.

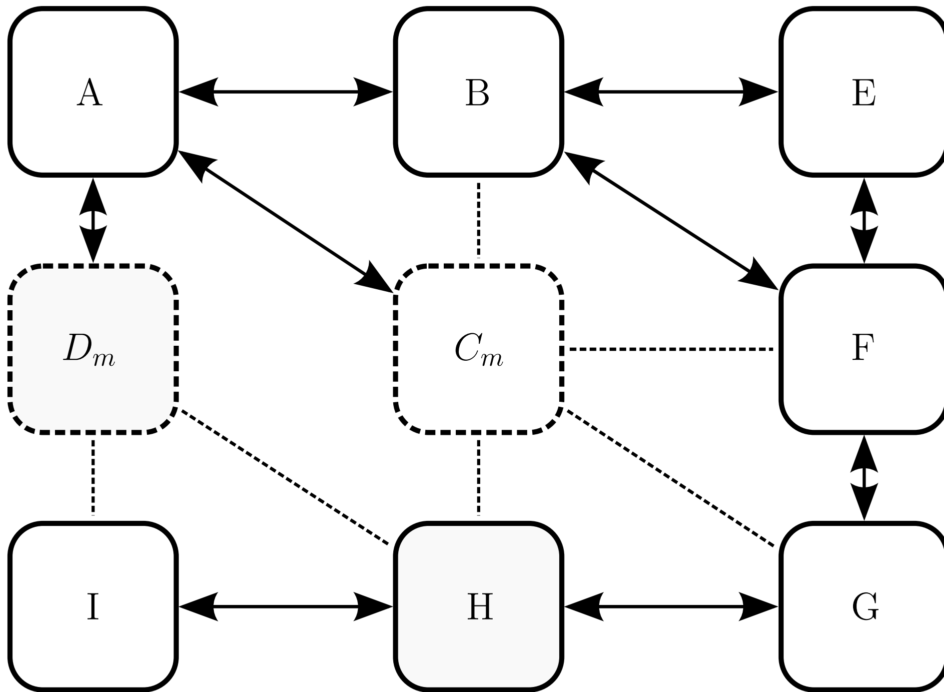


Figure 3.1: Denial of Service

3.1.2 Man-In-The-Middle attack

An adversary A could also try to execute a Man-In-The-Middle attack by intercepting the packages exchanged between two peers. To be able to conduct such an attack the adversary would first have to determine if two peers are direct neighbors or put differently if both peers are directly connected with each other. If that is not the case A would have to break anonymity

before being able to determine if the peers in question exchange any data at all. If the peers are directly connected A would have to correlate both peer ids with the transport protocols and corresponding addresses used for exchanging GNUnet packages between the two peers. Additionally A would need to have the means to intercept all those packages by controlling at least one router along the path.

Provided the adversary A can successfully intercept packets exchanged between two peers, he faces another problem. The content of the exchanged packets (messages) are AES encrypted on application level using keys only known to the two participating peers. Each incoming message is decrypted an re-encrypted with the AES session key also known to the recipient. Therefore the attacker would have to break AES encryption to be capable of reading the content. This would require access to a huge amount of processing power on the side of the attacker and is therefore highly unlikely.

If A actually succeeds in breaking the AES encryption and is able to inspect packages he will either see queries with an inexpressive resource id or replies with ECRS encoded content.

3.2 Authentication and Confidentiality

To directly conquer confidentiality in GNUnet, the attacker would either have to break the 256bit AES encryption of regular GNUnet messages or the 2048bit RSA encryption that is used for exchanging the AES keys. As far as the author is aware, no attacks, that work with acceptable time constraints, against these ciphers have been published until today.

Since encryption cannot be broken, the attacker is left with only one option.

To decrypt the messages exchanged between to peers, the attacker has to acquire the used encryption keys.

3.2.1 Rewrite Attack

To conclude a *rewrite attack* an adversary changes parts of the data contained in an encrypted message. If the used encryption protocol is mainly based on bit-wise additions or similar operations the attacker could be able to predict the changes in the decrypted plain text message and has therefore the means to forge message contents.

This attack cannot succeed against GNUnet, because of two reasons. GNUnet uses CFB mode for AES encryption, which is enough to prevent the attack according to [32]. Secondly the content of GNUnet messages is always supplemented with the hash value of itself. This allows peers to immediately determine if a message has been tampered with or not. Additionally contents are signed by the sender of the message providing further protection against forging of messages.

3.2.2 Reflection attack

The reflection attack as laid out in [33] uses interception and repeated sending of authentication messages to obtain a fully authenticated connection to a target. The attack is commonly executed in following manner. First the attacker initiates a connection to the target. The target then responds to the attacker by sending an authentication message containing a challenge. Next the attacker opens another connection to the target and sends the previously received challenge as its own. The target now responds to the challenge on

the new connection, hereby providing the attacker with the means to answer back on the original connection. If the attack is successful the target accepts the challenge received over the original connection and authenticates the attacker.

GNUnet prevents this attack by using different keys for sending and receiving of messages. Therefore if node A receives a challenge from another node B the message cannot be used as challenge from A to B. Also each GNUnet messages contains a peer id that identifies the sending (or indirecting) node. Therefore a reflected message would contain the id of the original sender, which is not supposed to happen at all and therefore clearly indicates an attempt of forgery.

3.3 Anonymity

Since anonymity is the main purpose of GNUnet, it is of course one of the first features to be subject to an attack. To successfully break the anonymity of GNUnet an attacker R has to determine if a query has been initiated by or a reply originated the attacked peer S. Therefore the attacker has to learn about the topology of the network and has to determine which path a message has taken.

As described in 2.3.1 on page 23 a GNUnet peer indirects queries it receives by stripping the sender's peer id from messages it receives and replacing it with its own, as long as the peer's CPU and bandwidth load is below a certain margin. The indirection mechanism prevents R from being able to learn the peer identity of the initiator of the query directly from the query itself. If the peer's load exceeds the above-mentioned margin or a low level of anonymity

is desired a peer will leave the peer id of incoming queries unchanged or in case of extreme loads it will even drop incoming queries. Thus the degree of anonymity deg_S, described in 2.3.1 on page 27, decreases.

Each of the attacks proposed against anonymity in GNUnet, which are mentioned in this thesis, requires the attacker to analyze the anonymity sets (see 3.3.1 beneath) of one or several peers over a certain period of time. The results of the analysis are then used to deduce which of the inspected queries where initiated by the peer under scrutiny S. If such a deduction can be made at all and with enough certainty depends on the deg_S, the resources available to R, including the number of peers, routers, available network bandwidth and processing power, and the number of observed anonymity sets.

3.3.1 Anonymity sets

The anonymity set of one round of anonymous communication by a mix node S consists of a set of senders of messages received by the node, the sender anonymity set, and the set of recipients which the node will send those messages to in the next step. Provided S receives a set of messages in a time interval t from a subset A' of all possible senders A. A' is then referred to as the sender anonymity set (illustrated in Fig. 3.2). The size of A' is $|A'| = a$ and one has $1 \leq a \ll |A|$.

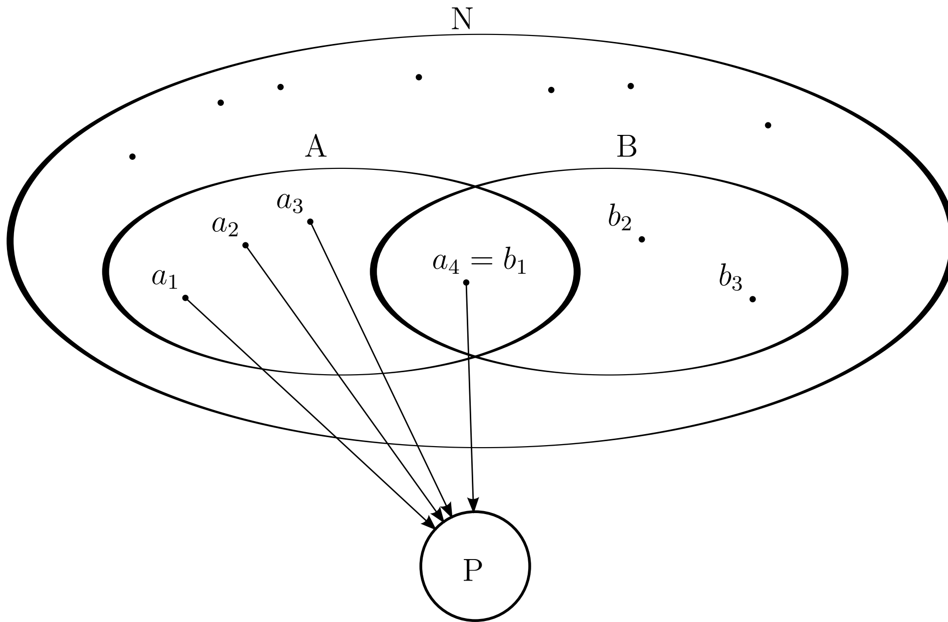


Figure 3.2: Sender anonymity set

In contrast $B^{'}$ is the subset of the possible recipients B that S will send the previously received messages to in the next time interval $t+1$. $B^{'}$ is referred to as the recipient anonymity set (illustrated in Fig. 3.3). The size of $|B^{'}$ is $|B^{'}| = b$ where $1 \leq b \ll |B|$ and $b \leq a$ and corresponds to the batch size of the mix S. It is to be noted that $A^{'}$ can contain elements of $B^{'}$ and vice versa, $A^{'}$ and $B^{'}$ can also be identical.

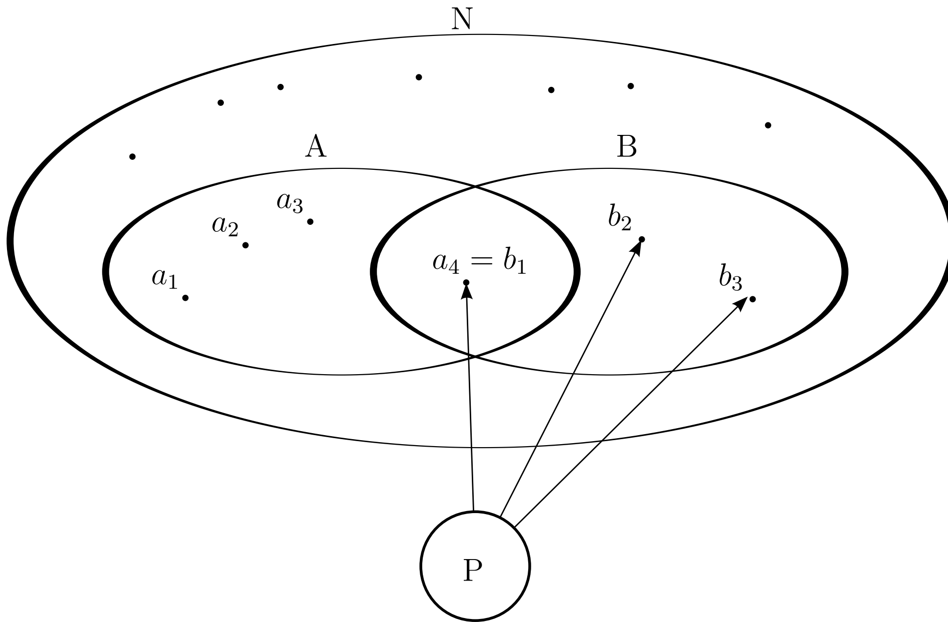


Figure 3.3: Recipient anonymity set

The union of these two sets constitutes the anonymity set of size m that characterizes the anonymous communication taking place between senders in the interval t and recipients in the interval $t + 1$. The anonymity set is the basic model for analysis for all attacks described in this section.

3.3.2 Probabilistic attacks

During a probabilistic attack the adversary tries to determine the communication partners of a peer by observing the anonymity sets resulting from network activity.

Disclosure Attack

The disclosure attack as laid out in [1] is the first probabilistic attack that will be discussed here. Its goal is to distinguish the b recipients B' of messages a

node S sends, from all possible recipients B in the network. The disclosure attack is subdivided into two phases, the learning phase and the exclusion phase.

Learning phase To be able to successfully complete the learning phase an attacker has to determine the size of the recipient anonymity set b of a node S. If the attacker can make enough observations, b can be determined recursively. If the learning phase could not be finished successfully, the attacker has to adjust his estimate for b and repeat the learning phase. Because of this fact, the assumption that the attacker can correctly determine b is considered valid.

During the learning phase an attacker tries to find b mutually disjoint recipient sets $\hat{B}$ from the messages S sends. For convenience the time t is increased each time S sends a message and the recipient set at time t is denoted as $\hat{B}_t = \hat{b}_{t_1}, \ldots, \hat{b}_{t_k}$. With this notation mutually disjoint recipient sets at time t can be defined as $(\hat{B}_{j_1}, \ldots, \hat{B}_{j_m}), \hat{B}_{j_x} \cap \hat{B}_{j_y} = \emptyset$ if $x \neq y$. If the attacker has found the b recipient sets at the end of the learning phase he can be sure that each set $\hat{B}_{j_x}$ contains only one of the peer communication partners of S.

Excluding phase Now the attacker observes further outgoing message of S and the resulting recipient anonymity sets. The goal of this observation is to exclude the non-peer partners from the disjoint recipient sets $\hat{B}$ found in the learning phase. It is achieved by intersecting new recipient set $\tilde{B}_l$ with the sets in $\hat{B}$ so that only one element is contained in the intersection. Put formally the refining of the basic sets $(\hat{B}_{j_1}, \ldots, \hat{B}_{j_m})$ requires the attacker to the use new recipient sets $\tilde{B}_l$ that only intersect with one of the basic sets

or $\tilde{B}_l \cap \hat{B}_{j_x} \neq \emptyset$ and $\tilde{B}_l \cap \hat{B}_{j_y} = \emptyset$ for all $x \neq y$. This is repeated until each element of $\hat{B}$ just contains one node, which means that all non-peer partners of S have been excluded from the basic sets. Therefore the remaining b users in $\hat{B}_{j_1}, \ldots, \hat{B}_{j_m}$ have to be the communication partners of S.

Discussion As already stated in [1] the disclosure attack is NP-complete and is equivalent to the well-known Clique problem [15]. Hence the attack does not scale with rising numbers of recipient sets and is therefore not applicable with reasonable cost against highly connected and busy nodes.

Statistical Disclosure Attack (SDA)/ Intersection attack

The statistical disclosure attack improves the disclosure attack described above, by introducing methods to reduce runtime and memory complexity. The attack uses the same model as the default disclosure attack in that a mix system is analyzed by observing the recipient anonymity sets. The process of analyzing the recipient sets however is the major difference to the original attack. Instead of executing costly intersection operations, probability values are applied to identify potential recipients of the node under scrutiny.

To analyze the recipient sets a vector $\vec{v}$ is defined with size $|B|$, containing one element for each of the potential recipients. Each element of $\vec{v}$ represents the probability if the corresponding node is to be the recipient of a message send by S. Assuming that an attacker R can only predict if a node can be recipient or not and has no further knowledge about the probability distribution, each element of $\vec{v}$ can be initialized with one of two values. If a node is not a recipient at all the corresponding element of $\vec{v}$ is set to zero. Otherwise it is assumed that the probability for possible recipients is evenly distributed and

thus the element of $\vec{v}$ corresponding to the node is initialized with $\frac{1}{m}$.

A second vector $\vec{u}$, that represents the probability distribution for all other senders to select their recipients for each time interval, is defined. Again it is assumed that probability is distributed uniformly over all potential recipients B and accordingly all elements of $\vec{u}$ are set to $\frac{1}{|B|}$.

To conclude the attack, R has to observe the messages that S sends in t intervals (as mentioned in 3.3.2 t is increased every time S sends a message, so the number of messages sent is equal to the index of the time interval). From these observations the attacker can compile a list of vectors $\vec{o_1}, \ldots, \vec{o_t}$ each of which represent the probability distribution of the recipient set of the message sent by S at the time corresponding time index. For a sufficiently large number t of observations one has:

$$\bar{O} = \frac{\sum_{i=1\ldots t} \vec{o_i}}{t} = \frac{\vec{v} + (b-1)\vec{u}}{b} \tag{3.1}$$

By using the observations $\vec{o_1}, \ldots, \vec{o_t}$, the batch size b of S and the probability distribution $\vec{u}$ for other senders, R can therefore calculate $\vec{v}$ as follows:

$$\vec{v} = b\frac{\sum_{i=1\ldots t} \vec{o_i}}{t} - (b-1)\vec{u} \tag{3.2}$$

Finally R multiplies each element of $\vec{v}$ with each element of observation $\vec{o_i}$ at time index i. The resulting vector is then normalized to yield $\vec{r_k}$:

$$\vec{r_k} = \frac{\vec{v} \cdot \vec{o_k}}{|\vec{v} \cdot \vec{o_k}|} \tag{3.3}$$

The elements of now $\vec{r_k}$ represent the probabilities of each corresponding node to be the recipient of the message sent by S at time index i. Hereby a

larger number of observations allows R to estimate probabilities with greater significance.

Discussion The statistical disclosure attack improves the disclosure attack by reducing the the computational complexity of the original disclosure attack. However the quality of the prediction is depending upon the number of observations and can therefore get quite expensive in large networks with highly connected peers. For example, to gain knowledge about the topology of a GNUnet network the attack would have to be executed against multiple peers along a path simultaneously to effectively track a specific message or message set.

Nevertheless, the concept of the statistical disclosure attack is very flexible and can be modified to suit many different implementations of mix-based anonymous networks, which the attacks extending or modifying the statistical disclosure attack described in [11], [24] or [23] eloquently illustrate.

Shortcut attack

The last probabilistic attack that will be described here is proposed in [22]. The attack is called *shortcut attack* and builds on the aforementioned intersection attack 3.3.2. Contrary to all previous probabilistic attacks the shortcut attack was especially tailored towards GNUnet. Therefore the adversary is supposed to operate a GNUnet peer himself. Hence the attacker does not monitor traffic of other peers at transport level, but analyses the ingoing packages of the peer he operates. [22] describes a basic and an improved variant of the attack. Since the basic attack is not applicable, as stated by the author, only the improved variant will be discussed here.

To he attack works under the assumption that linkable queries are received with a higher probability from the node (or with the peer id of the node) that is closer to the initiator, formally $P_A > P_B$ if peer A is closer to the initiator of the initiator than B. This assumption is based on the fact, that busy GNUnet peers stop replacing peer ids of incoming queries with their own. This implies, that more queries with the id of peer A arrive at the node of the attacker in a certain interval, than queries with the id of peer B.

Relying on this precondition the attacker subsequently tries to find the originator of a set of linkable queries. Therefore the attacker has to connect to all neighbors of the tested node and wait for linkable queries. Linkable queries relate to the same content or more precisely to the same set of IBlocks and DBlocks of a file. Then the attacker uses a statistical test to determine the neighbor that is closer to the initiator.

To test the neighbors $N_1, \ldots, N_n$ of the current node C it has to be decided for each pair (C, N_i) whether both nodes are equally likable to receive linkable queries from the originator or not. Since this examination is expensive to compute, because a lot of observations have to be analyzed, the attacker might first fall back to a simplified approach and relax the initial precondition from $P_A > P_B$ to $P_A \gg P_B$. That means the attacker estimates, that the probability to receive a query from the peer A closer to the initiator is much higher than the probability to receive a linkable query from B. Consequently the attacker now assumes, that the first query he receives is from the node that is closer to the originator. To verify this conclusion the attacker must again make enough observations, so that the hypothesis $P_A \neq P_B$ can be accepted with a sufficient significance. If a closer node exists, it is selected as the new current node and its neighbors have to be tested again, otherwise the current node must be the initiator.

The shortcut attack has the advantage over the other discussed probabilistic attacks, that it focuses on actual features of GNUnet and is not based on the usage of mixes. The lower degree of abstraction therefore could make the implementation of the attack easier for an adversary. On the other hand the approach suffers from some shortcomings, because [22] was published before the specification of GAP [7] was available and therefore not all information was available to the author. For example the size of GNUnet messages or packets is estimated as 1kb, whereas it actually is 32kb. This fact alone would multiply the size of data to analyze by factor 32, since the number of necessary observations does not change. Especially the friend-to-friend mode of GNUnet, which has not been considered by the author of [22], could make this attack completely infeasible.

Suppose an attacker has recursively processed several nodes to get closer to the initiator. Now he tries to connect to all neighbors of the current node. The neighbor that actually is closer to the originator of the query is operating in friend-to-friend mode and the peer id of the attacker is not contained in this peers friend-list file. Now the attacker basically has hit a wall with his efforts, because he cannot connect to the node in question and therefore cannot proceed with the recursive convergence to the originator.

3.3.3 Other attacks on anonymity

Besides probabilistic attacks there are other approaches to break an anonymity system. Basically there are two ways for an attacker to conclude a non-probabilistic attack. One option is to manipulate the recipient sets of one or more peers, as described in 3.3.3. The second alternative would be to gain additional information about an anonymous communication by observing

additional properties of the system, as in 3.3.3 for example.

(n-1) attack / flooding attack

The basic idea of the *flooding attack* is to send a big amount of messages to a peer, to force it to almost exclusively forward these messages. Suppose that a peer P receives n messages at a certain point in time and $n-1$ of these messages where sent by the attacker A. When P forwards the n messages A might now be able to identify its own messages and therefore also the single message M from another peer. In other words the attacker effectively reduces the size of the recipient anonymity set to one, thereby making it impossible for P to hide M.

The flooding attack cannot be applied against GNUnet for various reasons. The first reason is that the economic model hampers attempts of flooding nodes with queries by charging (reducing the local trust value) the sending node for forwarding its queries. Secondly the peer-to-peer nature of GNUnet makes it very unlikely, that an attacker can ensure that all messages bar one arriving at a peer P originate from him, since other peers most probably will also send messages to P. Finally messages sent between GNUnet peers are padded to uniform size and are AES encrypted and therefore the attacker is hard put to distinguish his messages from others.

Timing Attack

To identify messages indirected by a peer, an attacker could try to measure the time spans, required to execute encryption operations. Therefore an attacker A would have to observe the network traffic and must be able to

identify GNUnet messages. A connects to a peer P and sends a number of n messages to this peer. The messages A sends to P should have different sizes, so that A can expect that encryption operations will need different amounts of time. The attacker then measures the time between dispatching of the message and forwarding of the message by P to its other neighbors. If A can conduct enough time measurements he might be able to deduce which of his messages has been forwarded to specific neighbors. Thus the attacker A could enable himself to reconstruct paths messages take through the network, if A can execute this attack successfully against a sequence of peers. A more detailed view on timing attacks, including a more formalized specification, can be found in [20].

Several of the security techniques, GNUnet makes use of, provide protection against timing attacks:

1. mix concepts (batching, reordering)
2. delayed sending of messages
3. resending of messages

Each of these features adulterates the timing measurements concluded by A. Therefore timing attacks against GNUnet do not promise to yield exploitable results for an attacker.

3.4 Economy

Suppose an attacker A wants to flood a peer P with messages, to reduce the anonymity set of P. He is hindered in this approach by the economic model GNUnet makes use of. Remember that a P reduces the local trust-value

of its counterpart Q when forwarding messages sent by Q. A could try to circumvent this mechanism by executing a special attack, described beneath, thereby enabling himself to execute the flooding attack afterwards.

3.4.1 Sybil attack

The *sybil attack* as described in [12] is used to enable A to pose under multiple identities. This means the attacker appears as a set of unrelated participants. Subsequently A uses the increased number of participants under his control to mount other attacks.

In GNUnet it is not even necessary to execute the sybil attack because any user can run as many peers, and therefore can have as many ids, as he likes. As a consequence an attacker running a large number of peers could try to connect a specific peer P and send it a large number of queries to execute a flooding attack. GNUnet renders this approach futile by using the GAP protocol in combination with its economic model. A GNUnet peer, when busy, will not forward queries from neighbors whose local trust-value is not sufficiently high. Since newly connected peers begin with a local trust-value of zero an attacker can only expect that his messages will be forwarded as long as the peer is idle. When the attacker sends enough queries to stress the peer above its busyness threshold, P will stop forwarding messages from untrusted neighbors and therefore also messages originating from the peers of A. This effectively renders the increased numbers of A's peers useless for mounting any kind of flooding attack.

3.5 Deniability

An adversary that wants to execute censorship over intermediaries in GNUnet has to decide, if he wants to exercise editorial control over actual contents or if he wants to prevent searches for keywords associated with 'unwanted' content. Anyway the attacker subsequently has to work around the deniability provided by ECRS. The authors of [17] themselves have suggested possible attacks against ECRS, which are described hereafter.

3.5.1 Censoring queries

To execute censorship over queries an attacker, that has the means to force intermediaries to censor queries for specific contents, could try to compile a list of keywords under which these contents might have been published. From this list the attacker could generate the KBlocks related to the keywords and force intermediaries to drop queries requesting these blocks.

Although this approach allows an adversary to filter queries for certain popular or easy to guess keywords, it does not provide the means to censor specific content. Also this form of censorship is easy to circumvent by publishing content under multiple keywords. If content is published under a namespace the guessing of keywords alone would not even allow the attacker to construct the SBlock. To do so the attacker would have to additionally figure out the hash of the public key of the namespace. Therefore the censorship of queries for content in the global namespace will not affect the queries for content published within a namespace.

3.5.2 Censoring content

To force intermediaries to exercise editorial control (in the sense of identifying and blocking specific contents) an adversary has to be able to identify the encoded content that should be blocked (these content is referred to as suspicious content further on). To gain this ability it is necessary to identify and correlate the blocks resembling the file(s) in question, namely the DBlocks and IBlocks. To do so, an attacker has to search the network for keywords he associates with unwanted content. From each set of search results the attacker has to select those elements that point to content he deems to be suspicious. Finally the adversary has to download the selected content thereby obtaining the DBlocks and IBlocks.

Contrary to the sole guessing of keywords the strategy described above allows censoring specific contents and therefore constitutes a more subtle from of censorship. It is however much more expensive to apply, because contents have to be downloaded first and therefore memory demands grow linearly with the number of contents that should be censored. Again this attack is not hard to circumvent, since only minor changes to the content are necessary to obtain different IBlocks and DBlocks.

3.5.3 Flooding global keyword space

A destructive attacker could try to publish useless content under many different keywords and thereby polluting the global keyword space. If enough content is published the adversary could succeed in making searches for popular keywords return mostly valueless results.

The use of namespaces is a possible defense against this kind of attack, be-

cause only the owner of a namespace can use it to publish content. However peers then would have to decide if namespaces are owned by users that provide valuable content.

4 Conclusion

First of all it has to be noted, that the current version of GNUnet cannot be considered stable, since there are still major changes in progress, planned or in discussion. For example in one of the future versions GNUnet developers might switch from RSA encryption for exchanging session keys to ECC (Elliptic Curve Cryptography) [19]. This might cause further significant changes to the code and possibly the protocols of GNUnet. It also has to be noted, that previous versions are not protocol compatible with the current release or amongst themselves. Hence it can be expected, that still a lot of time and effort has to be spent by the authors of GNUnet, until a final release can be published.

Nevertheless, GNUnet is one of the most ambitious projects regarding the anonymization of a large-scale peer-to-peer application. Many commonly used security and anonymization techniques have been combined with features of previously existing anonymous networking applications to achieve anonymity in a large distributed peer-to-peer network.

The encoding for censorship resistant sharing (ECRS) as described in 2.5 extends the CHK-encoding introduced by Freenet [10] by using it on block-level rather than on file-level. This allows swarming, without impairing deniability at all.

GNUnet's anonymity protocol (GAP), as presented in 2.3, supplements classic mix techniques with automatic noise generation by content migration, thus achieving a higher level of anonymity.

RSA-encrypted exchange of session keys combined with a duplex handshake involving challenges and AES-link-encryption of messages provide for secure authentication and a high standard of confidentiality.

The economic model of GNUnet mainly uses the self-interest of peers to ensure, that contribution to the network is credited and usage of the network is charged for. Hereby protection against freeloaders or malicious peers trying to take advantage of a peers resources is achieved almost as a by-product.

All this combined yields the result that GNUnet proves to be resistant against a great range of attacks and allows users to conduct anonymous and secure file-sharing. It is to be hoped, that when a stable version finally becomes available and is included in the repositories of the major Linux distributions, GNUnet will become more widely used. This again would increase the amount of available content, further attracting additional users.

List of Figures

2.1 GNUnet layers [7] 9

2.2 Key exchange 14

2.3 Link encryption in GNUnet 20

2.4 Chaum mix 21

2.5 Source Rewriting 24

2.6 Indirection 25

2.7 Forwarding 26

2.8 Trust in a small GNUnet [**ebe:2003**] 31

2.9 IBlocks [17] 34

2.10 Tree of CHK-encoded blocks [17] 35

3.1 Denial of Service 39

3.2 Sender anonymity set (t) 44

3.3 Recipient anonymity set (t+1) 45

B.1 General configuration 70

B.2 Network configuration 70

B.3 Transport configuration 71

B.4 File sharing configuration 71

B.5 File sharing application 72

Bibliography

[1] Dakshi Agrawal and Dogan Kesdogan. "Measuring Anonymity: The Disclosure Attack." In: *IEEE Security and Privacy* 1.6 (Nov. 2003), pp. 27–34. ISSN: 1540-7993. DOI: 10.1109/MSECP.2003.1253565. URL: http://dx.doi.org/10.1109/MSECP.2003.1253565.

[2] Christian Grothoff et. al. *Build instructions for Debian 5.0 using Subversion*. 2012. URL: https://gnunet.org/installation-debian5-svn.

[3] Christian Grothoff et. al. *GNUNET_CRYPTO_RsaPrivateKey*. 2012. URL: https://gnunet.org/doxygen/d6/d16/struct_g_n_u_n_e_t___c_r_y_p_t_o___rsa_private_key.html.

[4] Christian Grothoff et. al. *GNUNET_CRYPTO_RsaPrivateKeyBinaryEncoded*. 2012. URL: https://gnunet.org/doxygen/d2/d33/struct_g_n_u_n_e_t___c_r_y_p_t_o___rsa_private_key_binary_encoded.html.

[5] Christian Grothoff et. al. *GNUNET_CRYPTO_RsaPublicKeyBinaryEncoded*. 2012. URL: https://gnunet.org/doxygen/de/d12/struct_g_n_u_n_e_t___c_r_y_p_t_o___rsa_public_key_binary_encoded.html.

[6] Christian Grothoff et. al. *PingMessage*. 2012. URL: https://gnunet.org/doxygen/d3/dd3/struct_ping_message.html.

[7] Krista Bennett and Christian Grothoff. "gap - Practical Anonymous Networking." In: *Designing Privacy Enhancing Technologies*. Springer-Verlag. Springer-Verlag, 2003, 141–160. URL: http://grothoff.org/christian/aff.pdf.

[8] Rajkumar Buyya et al. "Economic Models for Management of Resources in Peer-to-Peer and Grid Computing." In: Press, 2001.

[9] David L. Chaum. "Untraceable electronic mail, return addresses, and digital pseudonyms." In: *Commun. ACM* 24.2 (Feb. 1981), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/358549.358563. URL: http://doi.acm.org/10.1145/358549.358563.

[10] Ian Clarke et al. "Freenet: A Distributed Anonymous Information Storage and Retrieval System." In: *Designing Privacy Enhancing Technologies, International Workshop on Design Issues in Anonymity and Unobservability, ,, Proceedings 2001*. Berkeley, CA, USA, July 2000, 46–66. URL: http://www.ecse.rpi.edu/Homepages/shivkuma/teaching/sp2001/readings/freenet.pdf.

[11] George Danezis et al. "Two-sided statistical disclosure attack." In: *In Privacy Enhancing Technologies*. 2007, pp. 30–44. URL: http://www.freehaven.net/anonbib/cache/danezis-pet2007.pdf.

[12] John R. Douceur. "The Sybil Attack." In: *IPTPS*. 2002, pp. 251–260.

[13] Ronaldo A. Ferreira, Christian Grothoff, and Paul Ruth. "A Transport Layer Abstraction for Peer-to-Peer Networks." In: *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*

(GRID 2003). IEEE Computer Society. IEEE Computer Society, 2003, 398–403. URL: http://grothoff.org/christian/transport.pdf.

[14] FIPS. *Advanced Encryption Standard (AES)*. FIPS PUB. pub-NIST. pub-NIST:adr, Nov. 26, 2001, pp. iv + 47. URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979. ISBN: 0716710447.

[16] Christian Grothoff. "An Excess-Based Economic Model for Resource Allocation in Peer-to-Peer Networks." In: *Wirtschaftsinformatik* 3-2003 (June 2003). URL: http://grothoff.org/christian/ebe.pdf.

[17] Christian Grothoff et al. *An encoding for censorship-resistant sharing*. 2009. URL: https://gnunet.org/sites/default/files/ecrs.pdf.

[18] J. Jonsson and B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. RFC 3447 (Informational). Internet Engineering Task Force, Feb. 2003. URL: http://www.ietf.org/rfc/rfc3447.txt.

[19] N. Koblitz. "Elliptic curve cryptosystems." In: *Mathematics of computation* 48.177 (1987), pp. 203–209.

[20] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems." In: *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '96. London, UK, UK: Springer-Verlag, 1996, pp. 104–113. ISBN: 3-540-61512-1. URL: http://dl.acm.org/citation.cfm?id=646761.706156.

[21] H. Krawczyk and P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869 (Informational). Internet Engineering Task Force, May 2010. URL: `http://www.ietf.org/rfc/rfc5869.txt`.

[22] Dennis Kügler. "An Analysis of GNUnet and the Implications for Anonymous, Censorship-Resistant Networks." In: *Proceedings of Privacy Enhancing Technologies workshop (PET 2003)*. Ed. by Roger Dingledine. Springer-Verlag, LNCS 2760, Mar. 2003, pp. 161–176. URL: `http://www.freehaven.net/anonbib/cache/kugler:pet2003.pdf`.

[23] Nayantara Mallesh and Matthew K. Wright. "The Reverse Statistical Disclosure Attack." In: *Information Hiding*. 2010, pp. 221–234.

[24] Nick Mathewson and Roger Dingledine. "Practical Traffic Analysis: Extending and Resisting Statistical Disclosure." In: *In Proceedings of Privacy Enhancing Technologies workshop (PET 2004), LNCS*. 2004, pp. 17–34.

[25] Ralph C. Merkle. "A certified digital signature." In: *Proceedings on Advances in cryptology*. CRYPTO '89. Santa Barbara, California, United States: Springer-Verlag New York, Inc., 1989, pp. 218–238. ISBN: 0-387-97317-6. URL: `http://dl.acm.org/citation.cfm?id=118209.118230`.

[26] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: `http://www.bitcoin.org/bitcoin.pdf`.

[27] F. Reid and M. Harrigan. "An Analysis of Anonymity in the Bitcoin System." In: *Privacy, security, risk and trust (passat), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom)*. Oct. 2011, pp. 1318 –1326. DOI:

10.1109/PASSAT/SocialCom.2011.79. URL: http://people.scs.carleton.ca/~clark/biblio/bitcoin/Reid%202011.pdf.

[28] R. Rivest. *S-Expressions*. Internet Engineering Task Force, May 1997. URL: http://people.csail.mit.edu/rivest/Sexp.txt.

[29] R.L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Communications of the ACM* 21 (1978), pp. 120–126.

[30] Antony Rowstron and Peter Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems." In: *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Springer-Verlag. London, UK: Springer-Verlag, 2001, 329–350. ISBN: 3-540-42800-3. URL: http://portal.acm.org/citation.cfm?id=697650$\#$.

[31] R. Shirey. *Internet Security Glossary, Version 2*. RFC 4949 (Informational). Internet Engineering Task Force, Aug. 2007. URL: http://www.ietf.org/rfc/rfc4949.txt.

[32] Richard E. Smith. *Elementary Information Security*. 2011.

[33] Andrew S. Tanenbaum. "Computer Networks." In: 4th. Upper Saddle River, NJ, USA: PTR Prentice-Hall, Aug. 2002. Chap. 8, pp. 787–790. ISBN: 0-13-038488-7.

[34] Moritz Schulte Werner Koch. *The Libgcrypt Reference Manual*. 1.6.0-git5e14de0. Technical Report. 2011. URL: http://www.gnupg.org/documentation/manuals/gcrypt/gcrypt.pdf.

A Installation

Version 0.9.3 is the latest release of *GNUnet*. It is using the latest version of ECRS (ECRS v2). This section describes three ways, how *GNUnet* can be installed and configured on Debian 6.0.4 (squeeze).

A.1 Installation using release versions

In order to get a stable installation, tarballs of specific releases are used instead of the latest svn revision. The installation instructions for Debian 5.0 at [2] can be used, with slight modifications.

Installation of version 0.6.3 of libextractor:

Replace the command

```
svn checkout https://gnunet.org/svn/Extractor
```

with

```
wget -O ftpmirror.gnu.org/libextractor/libextractor-0.6.3.tar.
   gz | tar zxv
```

to download and extract the tarball for libextractor.

Installation of version 0.9.22 of libmicrohttpd:

Replace the command

```
svn co https://gnunet.org/svn/libmicrohttpd
```

with

```
wget -O -ftp://ftp.gnu.org/gnu/libmicrohttpd/libmicrohttpd
   -0.9.22.tar.gz | tar zxv
```

to download and extract the tarball for libmicrohttpd.

Installation of version 0.9.3 of *GNUnet*:

Replace the command

```
svn checkout https://gnunet.org/svn/gnunet
```

with

```
wget -O - ftp://ftp.gnu.org/gnu/gnunet/gnunet-0.9.3.tar.gz |
   tar zxv
```

to download and extract the tarball for *GNUnet*.

Installation of version 0.9.3 of *GNUnet*-gtk:

Replace the command

```
svn checkout https://gnunet.org/svn/gnunet-gtk
```

with

```
wget -O - ftp://ftp.gnu.org/gnu/gnunet/gnunet-gtk-0.9.3.tar.gz
   | tar zxv
```

to download and extract the tarball for *GNUnet*-gtk.

GNUnet-gtk dependencies not mentioned in the install scripts on `https://gnunet.org` are:

```
libgtk2.0-dev
libgladeui-1-9
libgladeui-1-dev
```

These packages are in the Debian repository and can be installed with apt-get or the Synaptic packet manager.

A.2 Installation using latest svn revision

The installation instructions for Debian 5.0 at [2] can be used. The dependencies for gnunet-gtk mentioned above are also required.

B Configuration

It is necessary to tell the linker the location of the GNUnet libraries. Therefore one can either use the following command to add the location to the linker path temporariliy:

```
export LD_LIBRARY_PATH=/usr/local/lib
```

The second option would be to add the location of GNUnet binaries to */etc/ld.so.conf* permanently:

```
sudo sed -i '$ a\/usr/local/lib' /etc/ld.so.conf
sudo ldconfig
```

For configuration of *GNUnet*, use:

```
gnunet-setup
```

The following window will open and provides the options for configuration of GNUnet:

Figure B.1: General configuration

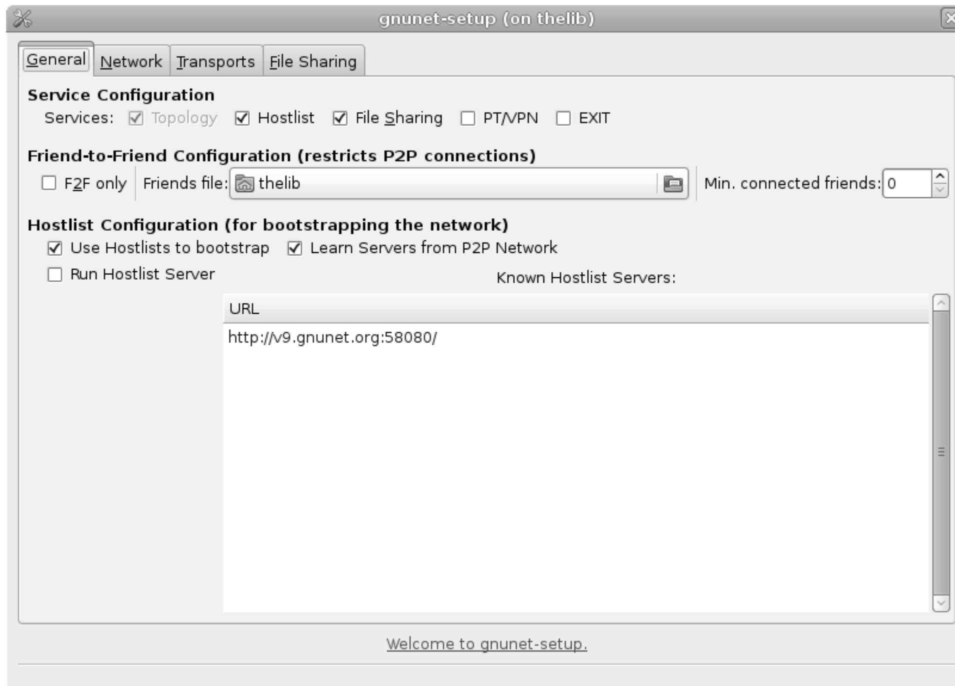


Figure B.2: Network configuration

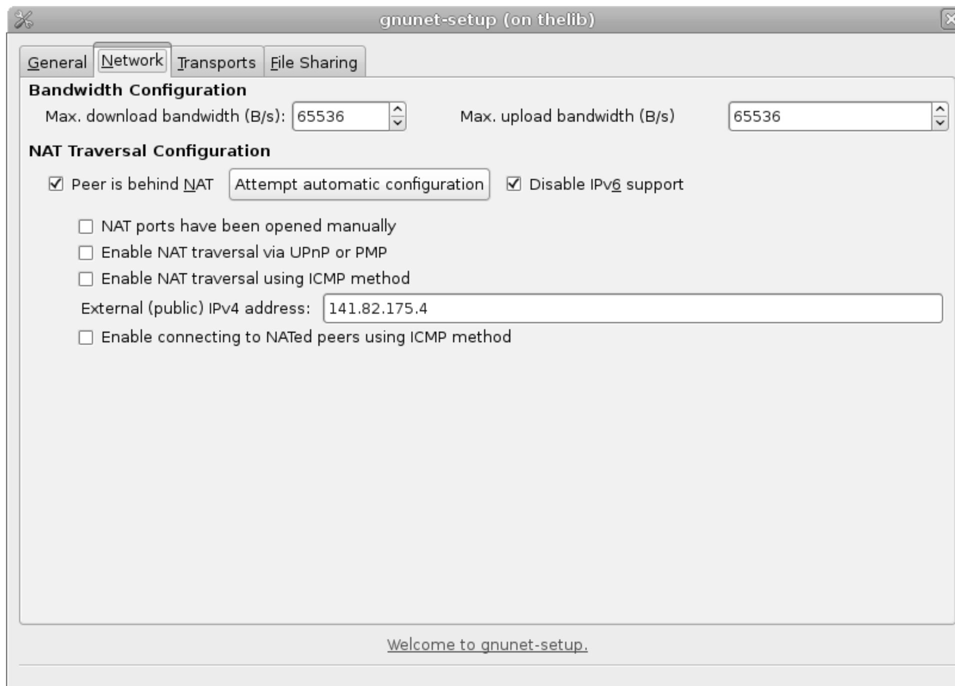


Figure B.3: Transport configuration

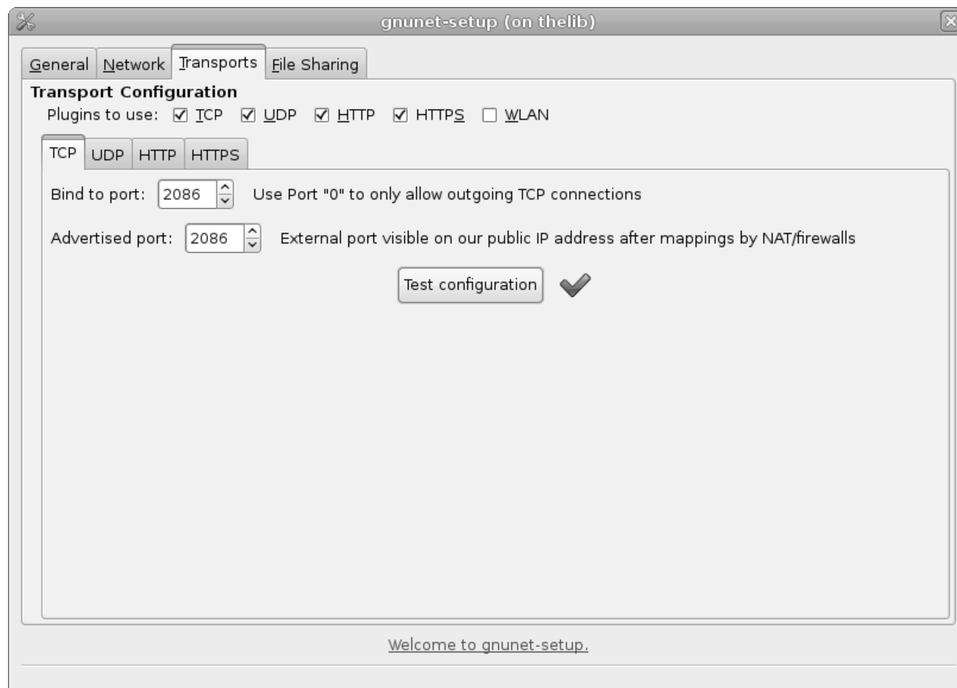
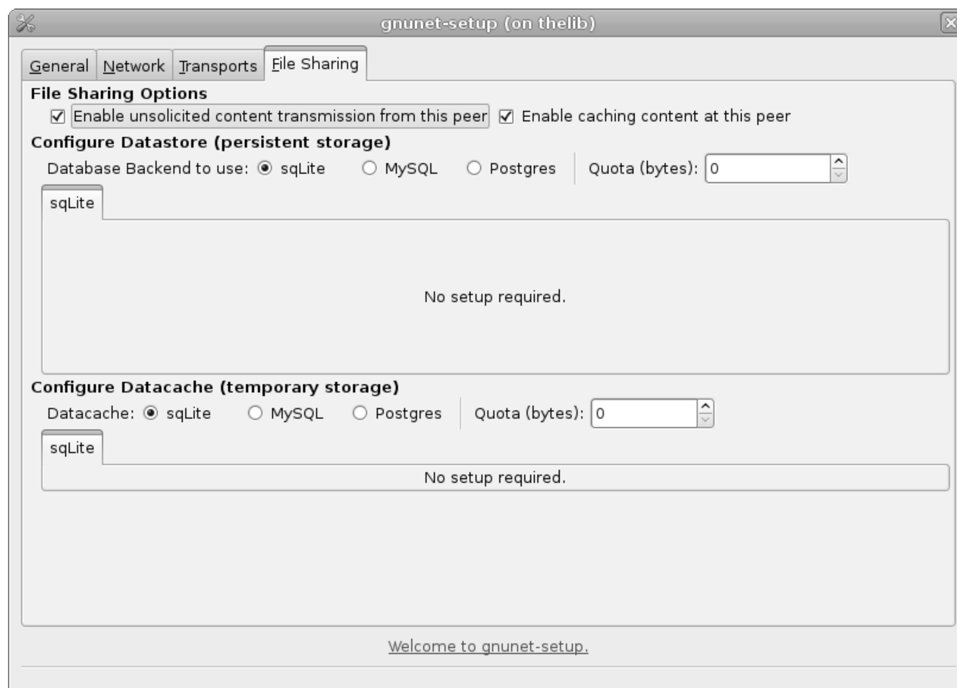


Figure B.4: File sharing configuration



Finally you can start the file-sharing application by using:

`gnunet-fs-gtk`

The following window will open and, as can be seen, allows searching, downloading and publishing.

Figure B.5: File sharing application