THESIS

ON THE DESIGN OF A SECURE AND ANONYMOUS PUBLISH-SUBSCRIBE SYSTEM

Submitted by

Dieudonne Mulamba Kadimbadimba

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2012

Master's Committee:

Advisor : Indrajit Ray

Indrakshi Ray
Leo Vijayasarathy

ABSTRACT

ON THE DESIGN OF A SECURE AND ANONYMOUS PUBLISH-SUBSCRIBE SYSTEM

The reliability and the high availability if data have made online servers very popular among single users or organizations like hospitals, insurance companies or administrations, this has lead to an increased dissemination of personal data on public servers. These online companies are increasingly adopting the publish-subscribe as a new model for storing and managing data on a distributed network. While bringing some real improvement in the way these online companies store and manage data in a dynamic and distributed environment, publish-subscribe is also bringing some new challenges of security and privacy. The centralization of personal data on public servers has raised citizens concerns about their privacy. Several security breaches involving the leakage of personal data are there to show us how crucial this issue has become. Significant amount of works has been done in the field of securing the publish-subscribe. However, all of these researches assume the server is a trusted entity, assumption that ignores the fact that this server can be honest but curious. This leads to a need for developing a mean to protect publishers and subscribers from the server curiosity. One solution to this problem could be to anonymize all communications involving publishers and subscribers. This solution will raise in turn another issue which involves how to allow a subscriber to query a file that was anonymously uploaded in the server by the publisher. In this work, we propose an implementation of the communication protocol that allows users to asynchronously and anonymously exchange messages and that also support a secure deletion of messages.

DEDICATION

To the Shining Star somewhere in the gallactic space who never stop praying for me

I lovingly dedicate this thesis to my parents, my relative, my host parents, my professors and my friends who supported me each step of the way

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Today a lot of peoples count on Internet companies to store their personal data, and to make them reliable and highly available through the internet. These online companies are increasingly adopting the publish-subscribe paradigm as a new model for storing and managing data on a distributed network. Distributed systems have been considerably changed by the tremendous development of the Internet which make them to include entities that are distributed all over the word. The variety of these entities and the dynamic nature of their behave have triggered the need for the development of more flexible communication models and systems. The publish-subscribe interaction scheme, a dedicated middleware infrastructure that loosely connect the different entities in those large-scale distributed systems, is receiving increasing attention [EFGK03].

The publish-subscribe is a communication model built around three sets of components that includes publishers, service providers, and subscribers. It is an event-based infrastructure in which subscribers are provides with the ability to express their interest in an event or a patterns of events that where produced by publishers. An event notification service is at the core of a publish-subscribe architecture. It represents a neutral mediator between publishers and subscribers, provides also the capability for storing and managing subscriptions while also providing an efficient delivery of events. Subscribers are consumers of events which call a subscrib() operation on the event service in order to register their interest in a particular event. They are no required to know the source of the events. The subscription information or interest expressed by the subscriber remains stored in the event service. Publishers are producers of information or events. The call the publish() operation on the event service in order to generate an event. The event service is respon-

sible to propagate those events to all subscribers based on the nature of their interest. A publisher can also use the advertise() operation in order to advertise the nature of their future events. It does so, on one hand, to help the event service to adjust itself to the expected flows of events, and on the other hand to let subscriber learn about new type of events that are available.

The adoption of the publish-subscribe model does not only improve the way Internet companies store data in a dynamic and distributed environment, but it also brings some new challenges of security and privacy [WCEW02]. One security issue is that the publisher of information cannot control the location of his data, since this later is stored and delivered across the Internet. Additionally, confidentiality and integrity of the data stored online is critical since publishers want to make sure that their messages are delivered only to relevant subscriber. However, online companies often fail to provide this guarantee. Thus, the dissemination of personal data on public server has raised a lot of concerns about the privacy of these data. Several famous incidents of personal data leakage were there to remind us this fact, and to raise our concerns. Several famous incidents of privacy violation originating from negligence, abusive use, internal or external attack, were there to remind us this fact, and to raise our concerns. For instance, in December 2009 RockYou!, a password database, was victim of a security breach that caused a leakage of information containing 32 million user names and plaintext passwords. In January 2009 a company named Heartland Payment Systems announced that it had been the victim of a security breach within its processing system. The intrusion has caused an estimated leakage of up to 100 million cards from more than 650 financial services companies. It had been called the largest criminal breach of card data ever. Most recently in 2010, a hacker pulled data from 100 million Facebook profiles and posted them online.

Furthermore, Event service requires subscribers to provide some personal information in order to verify them before granting them access to the services. This may incur threats of disclosures of personal privacy. Therefore we should provide a way to subscribers to perform an identity authentication on the Event service without explicitly disclosing the personal information to this later. This thesis proposes an implementation of a communication protocol that ensures an anonymous, asynchronous message exchange, and a message deletion. The proposed protocol combines the

advantage of diverse encryption schemes and the use of the Tor network to ensure a strong privacy and anonymity. The privacy is gotten by ensuring in one hand that all messages involved in the communication are encrypted, and in the other hand these messages, once stored in the server, carry no owner identifiers. Seemingly, the anonymity is gotten by ensuring that network nodes carry no identifiers.

Most of existing works on publish-subscribe systems have been largely focused on the problem of publish-subscribe performance and reliability. Only recently, some researchers start manifesting some interest to the security issues involving publish-subscribe systems. Wang et al [WCEW02] have been interested to the studies off specific security requirements of a content-based publish-subscribe system. In that paper, confidentiality, integrity and availability are identified as general security needs of a publish-subscribe application. That paper is limited to identifying the security problems, but does not present any solution to those problems. Significant amount of researches have been done in the field of securing a publish-subscribe [SL05] [SL07] [MN09]. However, all of these researches assume that the Event service is always a trusted entity. This assumption ignores the fact that this third party may be honest by curious. Thus, there is a necessity to protect the subscribers and the publishers against the curiosity of the Event service by, among others, anonymizing all the communications.

Considerable research efforts have been made to ensure anonymity of two or more users exchanging messages through a public server. These works include the Freenet Network, the Crowds network, and the Tor network. Freenet [CSWH01] is a peer-to-peer network that permits to both authors and readers the publication, replication, and retrieval of data while protecting their anonymity. Freenet is a location-independent and completely decentralized network. It is location-independent in the sens files are dynamically replicated in location near requestors and deleted from location where they are not interested. It is therefore almost impossible to locate the true origin or destination of a file. For the same reason a node cannot be held responsible of a file that it physically stores.

Crowds [RR98] is an anonymity network that hides actions of a single user within a crowd of multiple other users. Crowds protocol is designed to route each user's communication randomly in

a group of similar users, making it difficult to identify the true origin and the true destination of a particular message. This concept is what is called the users blending into a crowd of computers. It have been demonstrated that Crowds fails to protect the user again some specific attacks such that : the predecessor attack, the global attacker, or the local eavesdropper. Besides failing against some specific attacks, Crowds is not to protect the anonymity, and the privacy of a user who accidentally releases his identity into the server.

Tor [DMS04] is a privacy enhancing network that aims to protect internet users against a traffic analysis of their activities. His low latency make it ideal for activities such that web browsing, instant messaging, file sharing, and communication forms. Although it's a great in term of anonymity providing, Tor cannot protect Internet users activities against monitoring at the boundaries of the Tor network, where the traffic is entering and exiting the network.

Although these systems make the users internet experience more secure, they do not address the issue of privacy of information stored in a public server, mostly in case of leakage of information. The protocol that is implemented in this thesis try to tackle that issue.

Chapter 2 presents the related works to our protocol. We describe there the implementation of some networks whose protocol have inspired our work such as : The Freenet network and the Crowds Network. Since in this work we will be using the Tor network for anonymize our communication, in the chapter 3 we presents in more detail the Tor network. In chapter 4 we introduce some basic notions in cryptography to readers without prior knowledge of cryptographic concepts. The chapter 5 present the design of the protocol we are going to implement in this work. The challenges that we faced during the implementation of our protocol will be discussed in the chapter 6. The chapter 7 will provide a detailed descriptions of the major components of our implementation. Chapter 8 holds a discussion, a conclusion, as well as a proposal for the future development of this work.

Chapter 2

Related Works

Before we dive deeply into the description of our protocol, it would be wise to have an understanding of what is a publish-subscribe system as well as what are its security requirements. We will also give a look at some major protocols that drove the development of the field. This list, which is not exhaustive, includes protocols like : Ants protocol, The MIXes protocol, the Freenet, the Crowds protocol, and the Tor network as implementation of the Onion Routing protocol.

2.1 Publish-Subscribe system

2.1.1 Definition

The publish-subscribe is a communication model built around three sets of components that includes publishers, service providers, and subscribers. It is an event-based infrastructure in which subscribers are provides with the ability to express their interest in an event or a patterns of events that where produced by publishers. An event notification service is at the core of a publish-subscribe architecture. It represents a neutral mediator between publishers and subscribers, provides also the capability for storing and managing subscriptions while also providing an efficient delivery of events. Subscribers are consumers of events which call a subscrib() operation on the event service in order to register their interest in a particular event. They are no required to know the source of the events. The subscription information or interest expressed by the subscriber remains stored in the event service. Publishers are producers of information or events. They call the publish() operation on the event service in order to generate an event. The event service is respon-

sible to propagate those events to all subscribers based on the nature of their interest. A publisher can also use the advertise() operation in order to advertise the nature of their future events. It does so, on one hand, to help the event service to adjust itself to the expected flows of events, and on the other hand to let subscriber learn about new type of events that are available [EFGK03].

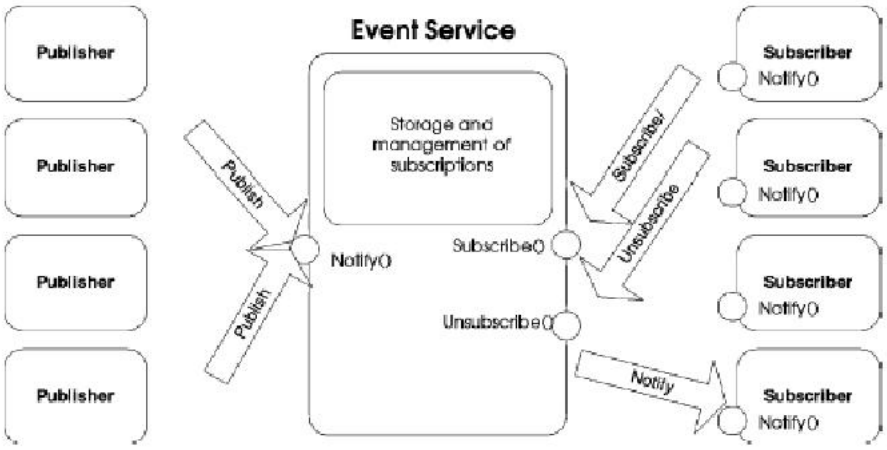

Figure 2.1: A simple object-based publish/subscribe system.

2.1.2 Message filtering

In the publish-subscribe model, filtering is the mechanism of selecting messages from publishers and processing them. In many publish-subscribe systems, publishers messages are delivered to an intermediary server called message broker to which subscribers register their subscriptions. The broker is responsible to perform the filtering which it does by using a store and forward function to transport publishers messages to subscribers. Subscribers receive only a subset of messages to which they had registered some interests. Two major forms of filtering are used inn publish-subscribe systems: topic-based and content-based [Pub10].

In a topic-based publish-subscribe system, messages are delivered to topics, with each topic corresponding to a named logical channel. In this system, subscribers are delivered messages that correspond to the topic to which they subscribed, and all subscribers to the same topic will receive the same messages. The classification of messages is done by the publisher.

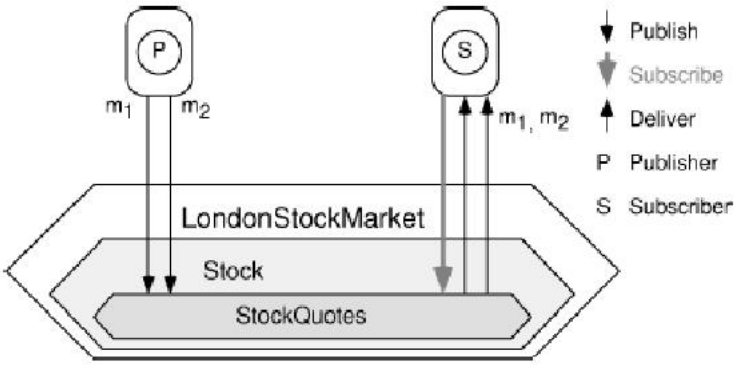

Figure 2.2: Topic-based publish-subscribe interactions.

In a content-based publish-subscribe system, subscribers receive only messages whose content corresponds to constraints defined into the subscriber interest message. The classification of messages is done by the subscribers.

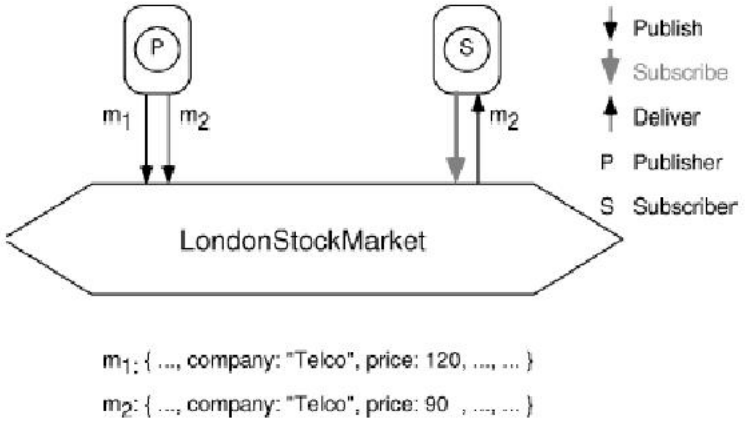

Figure 2.3: Content-based publish/subscribe interactions.

These two forms are the mainly used in publish-subscribe systems. There exists a filtering form that is a hybrid of the two main filtering forms. In this form, publishers post topic-based messages while subscribers express content-based interests to one or more topics.

2.1.3 Security requirements

One of the attractiveness factors for publish-subscribe systems is their capability to handle information dissemination across distinct authoritative domains, heterogeneous platforms and a dynamic environment of publishers and subscribers. This diversity of environments brings new security challenges [WCEW02]. In the scenario where Alice sends a message to Bob through a publish-subscribe system, we can define the key aspect of security as following. Authentication is the ability for Bob to be sure that Alice is the true sender of the message. In a publish-subscribe system, publishers and subscribers are not directly connected, and the communication between them is done in an asynchronous manner. The two parties are often anonymous to each other, mostly in a content-based publish-subscribe system. Therefore, it is challenging to make publishers and subscribers authenticate each other in this condition.

In traditional situation, confidentiality is the ability for Alice to prevent others than Bob from reading his message. Unlike in a traditional situation, in a publish-subscribe system, besides Alice and Bob, the server is also involved. Neither Alice or Bob have the control on the server, and therefore cannot prevent it from reading the messages since the server is needed for the routing of the message from Alice to Bob. So it is challenging to obtain a complete confidentiality. Integrity is the requirement that ensures that Bob received a message identical to the one sent by Alice. Due to the different nature of messages circulating in a publish-subscribe system, the integrity problem extends in three different forms: information integrity, subscription integrity, and service integrity [LP03].

Accountability is the ability for Bob to prove that only Alice could have sent the message. This problem arises in a publish-subscribe system because there is not direct relationship between publisher (Alice) and subscriber (Bob). Therefore no publisher can identify which subscribers can receive the message. The last requirement is the user anonymity. It is the ability for both publishers and subscribers to remain anonymous toward the server while exchanging messages. This anonymity can be achieved with various anonymizing techniques as the ones developed in the following sections of this chapter.

2.2 The Ants protocol

Ants protocol [GSB02] is an anonymous peer-to-peer protocol designed for a network in which nodes don't have a fixed position and don't reveal their true identity. The network is designed in the way allowing each user's node to use a pseudo identity instead of his real identity. A node realizes a search for a file on the network by broadcasting the search message.

The search message contains in its head the nodes pseudo identity, the message identifier, and a time-to live counter. The message identifier is unique for each message. The node neighbors, after receiving the search message, records the pseudo address of the node sender, as well as thee connection on which the message was transmitted. This node will in turn broadcast the message ti all of its neighbors. The process will repeat over and over at each receiving node until the message time-to live will expire. This protocol allow a decentralized routing because each node builds its routing table made of pseudo identities and the connection that has transmitted the most messages from each registered pseudo identity. A node can send a message to another node just by addressing the message to the other node pseudo identity [CC05].

There are anonymous systems that have been made based on the Ants protocol. We can list the Ants P2P, Mute, and Mantis [SM04].

2.3 MIXes protocol

MIXes [Cha81] is an anonymous peer-to-peer protocol that provides anonymity by forwarding messages from node to node with the particularity that the receiver node does not send immediately the message upon reception. It wait until it has received a certain number of messages that it will send after mixing them up. When done correctly, this message mixing can provide either sender anonymity, either receiver anonymity, or can hide sender-receiver linkability from an attacker able to see the whole network. The asymmetrical nature if downloading files makes it at time difficult for any node to collect enough messages in order to realize a good mixing. This is the drawback of the MIXes protocol [CC05].

Because of their original design, certain networks have received more attention from the re-

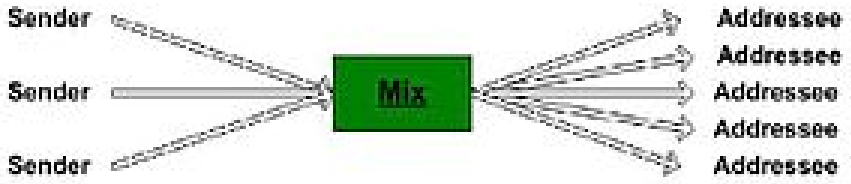

Figure 2.4: How MIXes network works

search community. We are talking about the Freenet network, the Crowds network, and the Tor network. We are going to describe them more in detail here.

2.4 Freenet network

Freenet [CSWH01] is a distributed peer-to-peer application which allow users to publish and obtain information one the Internet without fear of censorship. The network is built based on users individual computers. Freenet aims to guaranty the anonymity of both the producer and the consumer of information. It is intended to resist at deny of access to information, and to provide a dynamic storage and routing scheme, as well a decentralization of all network functions among participants computers [CC05].

2.4.1 Architecture

Freenet is a peer-to-peer network type whose nodes query each other in order to retrieve and store data files. The data files are queried according to a key that is location-independent. Freenet make use of storage space made available by each node as local datastore. The nodes also provide a dynamic routing table which is constituted by the addresses of other nodes and the keys that they hold.

To make a search a node query only its neighbors, since nodes are in chain in which they know only their immediate upstream and downstream neighbors. The generated query is assigned a hop-to live count, as well a pseudo random identifier. The hop-to-live, by decrementing at each node, allow the network to prevent an infinite loop. In addition, the pseudo-unique random identifier is

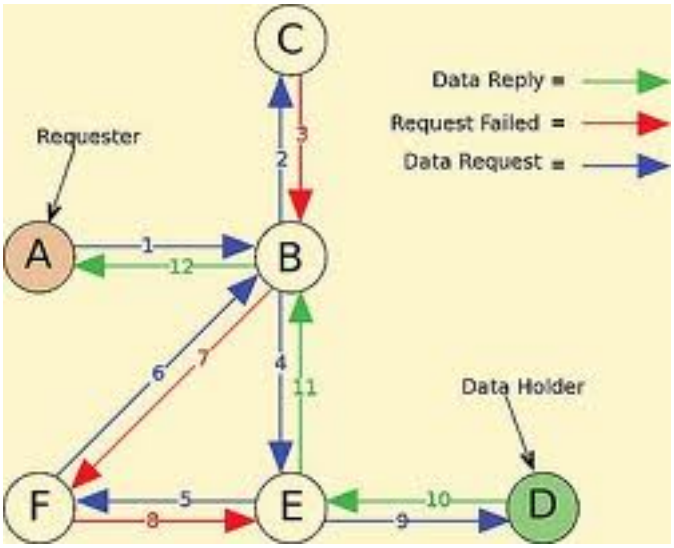

Figure 2.5: Freenet routing

used in a mechanism to prevent a loop, by making each node to reject queries that they have already processed. With this architecture in mind, how does a user to retrieve, to store, or to manage data?

2.4.2 Retrieving data

To retrieve data, the first element a user needs is the file key, which is obtained by hashing a short string describing the needed data. The user then generate a request constituted by the file key, and to which he assign a hops-to live value. The user then sends that request to his own node in order to check if his own local Datastore has the needed file. In case he does not have the file in his local storage, his node will propagate the request to his immediate neighbor, which in turn will check their own local storage for the file. In case one these nodes find the data, it will cache one copy and send it back to the requesting node, along with a message certifying that it's the source of that data. Otherwise the node will look in its routing table for the closest key entry to the request key, and then forward the query to the node corresponding to that key entry found in the routing table. This next node will in turn do the same process, and returns back the data to its requester node, which will send back the data to the original requestor node, but after caching a copy and after updating its own routing table with the key and the node source of the data.

There are some cases when a node can report a failure to the requestor of the data. One is when

the hops-to-live count of the request message expires. The other is when a node cannot forward the request to its successive best candidate nodes because they are down.

2.4.3 Storing data

Similarly to retrieving data, storing data starts with the user making a key file, key obtained by hashing a short string describing the data. The user then create an insert message constituted by the key file and a ho-to-live value assigned to the message. He then send that insert message to his own node, which in turn will perform a look up in his Datastore for the existence of that key. If the key already existed in the node, the node will announce a collision to the user by returning the file corresponding to the key. In this case the user needs to create another different key for the file he want to insert, and starts over the process.

When the node does not announce any collision to the user, that means the key was not found. In this case the node performs a look up in the routing table in order to find the closest key to the user key, whose corresponding node will be forwarded the insert message. This node in turn will try to insert the user key. If a collision is announced, the corresponding data will be returned, and the node send it back to the upstream node. This upstream node in turn will cache the data and update his routing table with the source of the data.

2.4.4 Managing data

The node uses the Least Recently User (LRU) algorithm to manage the stored data. When a new file needs to be inserted, the least used files will be recursively removed until there is enough room for the new file. The removal of these least used files has little effect on the availability of the files because the entry in the routing table will stay longer and can help to request the data from its source. The routing table data is also managed following the same principle.

2.5 Crowds

Crowds [RR98] is a protocol that provides anonymity to web-transaction. In this protocol, users who are willing to communicate anonymously with a web server are put into a group called

Crowd. The idea is to hide one user's actions into the actions of other users by letting these other users route each message on a randomly created path until reaching the web server. This is done in a way no to let either the web server, nor any node to guess the originator of the message [CC05].

2.5.1 Architecture

In order to understand how Crowd is designed, we need to understand theses two notions: The notion of the Jondo and the one of the blender. A Jondo is a process that an user can start on his computer, and that conveys a faceless representation of the user in the Crowd. A blender is the server that is managing the Crowd. A Crowd network is made of several users represented in the Crowd by their respective Jondos, and the blender.

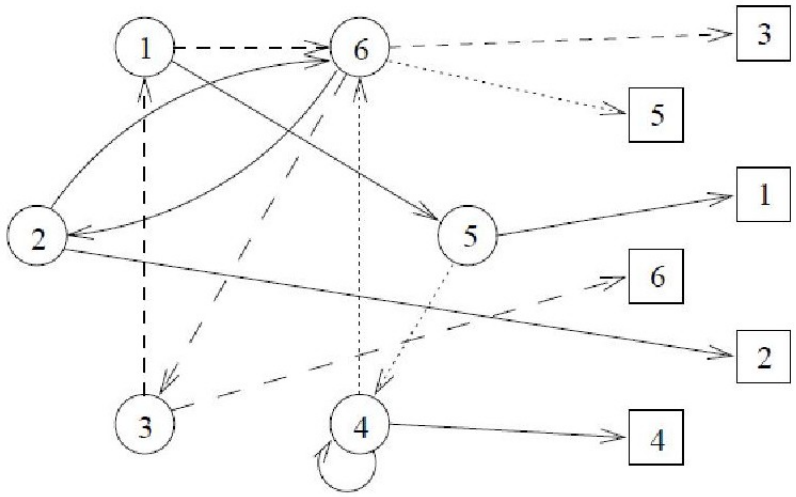

Figure 2.6: Crowds architecture

When an user want to join the Crowd, he starts the Jondo on his own computer and this Jondo will contact the blender to request a Crowd admittance. The blender will make the decision to admit that Jondo or not to the Crowd. If the decision is positive, the blender will send to that Jondo the information necessary to enable it to join the Crowd, as well the state of the membership of the Crowd at that time. Once admitted, the user will set his Jondo as his web proxy, and all his web request will be going through his Jondo.

When the user initiate a web request, the request is sent directly to his Jondo. The Jondo, upon

receiving he request, will forward the request to another Jondo randomly picked from the Crowd. This other Jondo uses a mechanism of flipping a coin in order to know if it should forward the request to another Jondo or just submit it to the end user. If the outcome of the coin flipping is in favor of the end user, the Jondo will submit the request to the end user, otherwise it will forward that request to another Jondo also picked randomly in the Crowd. So, in definitive a request travels on a path that goes from the user's browser to the end server, by crossing one Jondo least or a chain of Jondos. Knowing how the Crowd architecture is designed, one would wonder to know how user's membership is managed in the Crowd and what kind of security the Crowd provides to its users.

2.5.2 Membership Management

Crowd membership management consists in setting the mechanism and the policy that determines an user's admittance to the Crowd. Many schemes have been proposed for membership management, and the blender is the one in charge to perform that management. In this work we will describe a simple and centralized scheme for membership management.

When an user request an admission to the Crowd, the blender will request that user to create an user account and to provide a password. Those information will be stored by the blender. The stored password will be user for authenticating each communication between the blender and the user's Jondo every time an user starts a Jondo. The new user, upon admission in the Crowd, will be added to the members list by the blender and this list will sent back to the user by the blender in order to inform him about the state of membership within the Crowd.

Besides maintaining a members list, the blender maintains also a list of shared keys, each belonging to each user admitted in the Crowd. The existing Jondos in the Crowd are not aware of the existence of the new Jondo. So the blender will send them the new member information, as well his shared key. The existing Jondos will have to update their members list and their shared keys list. Any tow Jondo that are willing to communicate will exchange their keys, and these keys are know only by the two. Therefore the communication between two Jondo is encrypted. At this point we have to mention that each Jondo maintains its members list, and then is free to add or to

delete another Jondo from the members list. The deletion decision is taken when the incriminated Jondo failed.

The blender also provides some policy about allowing new member to join the Crowd. One important aspect of the policy involves to prevent any Crowd member from having more than one Jondo. The reason why is it important to reinforce such a policy is to avoid to have collaborative Jondos. Collaborative Jondos are Jondos that know each other, and can decide to coordinate their actions. Therefore they represent a threat to the Crowd security. Collaborative Jondos can be prevented from occurring by setting two different kinds of crowd. The first one is a relatively small Crowd made of people who knows each other. Since all members are known, it will be pretty easy to allocate to each of them an unique user account. This ensures that every member has only one Jondo in the Crowd. The other kind of Crowd network will be a public and larger network. This Crowd, by being large enough, offers a low probability of having a substantial part of Crowd who know each other, and therefore provides the necessary privacy.

2.5.3 Crowd security

The security of the Crowd have been tested against 3 kinds of attackers. A local eavesdropper, an end server, and against collaborative Crowd members. A local eavesdropper is an attacker who can sit on any particular computer that is part of the Crowd and then observe all outgoing and incoming communication to that computer. In the Crowd the end user Server is a web server. Therefore an end server attack is the one targeting the web server and listening the all communications from the user to the server. Collaborating Crowd members are the ones that agree on their own policy within the Crowd.

The analysis of the Crowd security shows that a eavesdropper can see that a message from the computer it is observing is not an answer to a previous incoming request. Therefore in this circumstance it breaks the anonymity of the sender. The analysis shows also that an end server attack is less dangerous because an attacker listening the traffic to and from the web server will not know which user is communicating with the server. The reason is that the request message from the user cross one or more Jondos and the communication is encrypted. In this case the only

anonymity that will be broken is the one for the server, but the server is public.

The last analyzed attack is the collaborating Jondos. Unlike the two previous attacks, this attack has thes Jondos as attackers. It's obvious that those collaborating Jondos know the server, but this is not what is important to them. Their objective is to identify the Jondo that has originated a request, and the only way they can do that is by guessing who is that Jondo. The probability of collaborating Jondos to make a correct guess depends on how many of collaborating Jondos there are and the total number of Jondos in the Crowd. One fact to notice is that even if a path goes through a chain of collaborating Jondos, the Jondo that is important in the chain is the first one, called the first collaborator. This Jondo is the one responsible of guessing the requestor Jondo. The other fact is that the first collaborator considers as his best guess for the requestor Jondo the Jondo that sent the message to him. The other non-collaborating Jondos could have be originator of the message, but the Jondo that sent the message to the first collaborator has the highest probability. Hence, for the first collaborator the question is what is the probability that the preceding Jondo is the actual originator of the request. If we consider N as the number of collaborators, P as the probability of forwarding a request to an another Jondo, and C as the number of collaborators, then Reiter and Rubin [RR98] establish a theorem that states that the preceding Jondo has a probable innocence, that is, it has no greater than 50% chance. Reiter and Rubin also provide the proof of the theorem in [RR98].

2.6 The Onion Routing protocol

The Onion Routing [RSG98] is a protocol designed to provide anonymity to users when using public networks. This anonymity is guaranteed to the user at the condition that he knows the public keys of all the intermediary nodes between him and the receiver. In order to send a message, the user first established an anonymous connection by creating a circuit through a number of randomly selected nodes, called Onion Routers. In order for that connection to take place, the initiator will access the Onion Routing network by mean of an application proxy that will format all the messages according to the Onion Routing network. The application proxy then connects to a Onion proxy that will construct an Onion. An onion is a data structure that is recursively layered, and

whose each layer is encrypted with the key of he corresponding Onion Router. The Onion holds the information necessary for the routing [CC05].

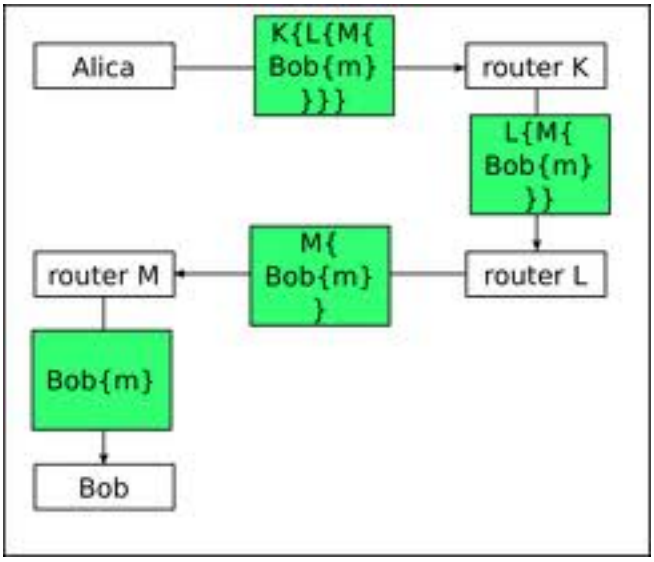

Figure 2.7: Onion Routing diagram

Chapter 3

TOR : The Onion Router Network

Tor [DMS04], standing for the Onion Router, is an implementation of the second generation of the Onion Routing protocol. Many limitations have been remarked in the first generation of the Onion Routing protocol. Indeed, it have been remarked that in the original Onion Routing protocol, among other issues, an user had the possibility to record traffics and could later use that information to compromise other nodes in the network. Another issue in the original design was the lack of a congestion control mechanism. The second generation of the protocol have been designed to address these issues and others.

3.1 Tor architecture

Tor, as an Onion Routing System, knows four phases. The first phase is the network setup. It's during this phase that an user initiate a longstanding connection between Onion Routers. The second phase is the connection setup, during which an anonymous connection is established through the Onion Router network. The third phase is the data movement, during which an user transmits or requests data through the anonymous connection set during the previous phases. The last phase is the destruction and cleaning up of the anonymous connection.

Understanding how Tor is built and how it works requires us to give a close look at its components and their functioning. When talking about using the Tor network you need : an Onion Proxy (OP), Onions, several Onion Routers, and one or several Directory servers.

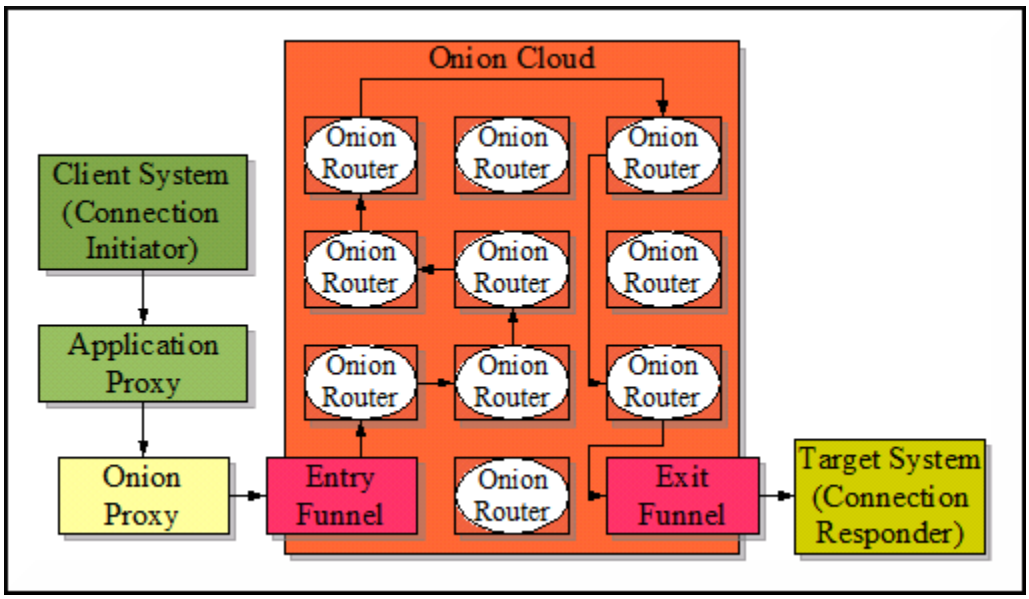

Figure 3.1: Tor architecture

3.1.1 The Onion Proxy

When an user makes a request and want to use the Tor network, the first interface on his way is the Onion Proxy. The Onion Proxy is a local software run by the user in order to access the Tor network. It's also called Tor Client. The Onion Proxy is needed to fetch directories, to establish circuits across the Tor network, and to handle connection coming from user application. Because it is a bridge between user applications and the Onion Routing network, the Onion Proxy must understand both application protocols and Onion Routing protocols.

As said above, one of the mission of the Onion Proxy is to construct circuits across the Tor network. Basically the Onion Proxy constructs circuits incrementally. The way it proceeds is by negotiating a symmetric key with each Onion Router (OR) that take part to the circuit, hop by hop. Unlike the first generation of the Onion Routing which build one circuit for each TCP stream, Tor enables many TCP streams to share the same circuit. An Onion Proxy can perform several actions in order to manage circuits. It can build as many circuits as needed, creating them preemptively to avoid delays. It can also expire a circuit which does not have any open stream. For security purpose, an Onion Proxy moves to a new circuit once a minute.

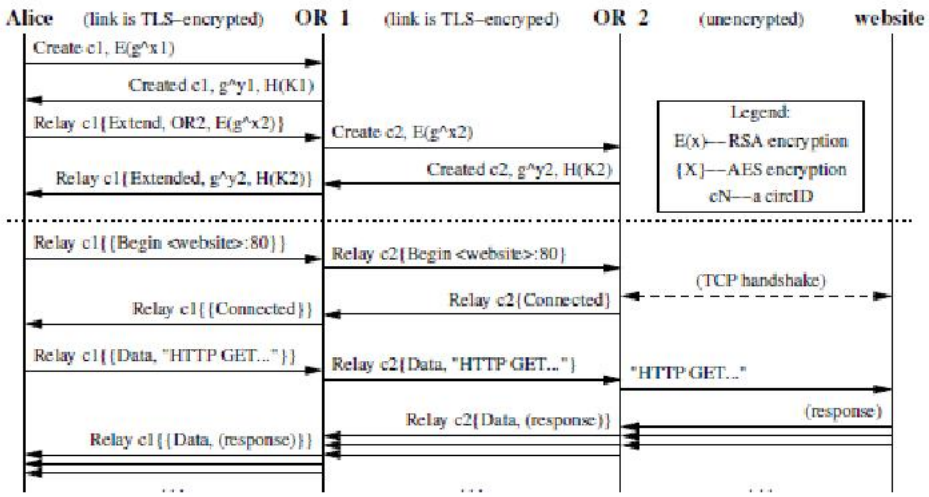

Figure 3.2: Alice build a two-hop circuit and begins fetching a web page

3.1.2 Onion

When building the anonymous connection, the Onion Proxy will create a data structure called an Onion. An Onion is a data structure that is recursively layered, and whose each layer is encrypted with the key of the corresponding Onion Router. It encapsulates the route information necessary to set the anonymous connection. The Onion Proxy creates the Onion iteratively, starting by the innermost layer. Each iteration correspond to the encryption of the Onion with the public key of the Onion Router which will need to read that layer. It does so by starting from the last Onion Router before the exit and working backward to the first Onion router in the circuit. By this way, the innermost layer of the Onion is intended to be decrypted by the last Onion Router in the circuit whereas the outermost layer is intended to be decrypted b the first Onion Router in the circuit.

3.1.3 Onion Router

Another major component of the Tor network is the Onion Router. Onion routers are individual computers that are used as nodes to constitute the Tor network. They help to relay data from the user initiator to the requested destination or vice versa. An user willing to let his computer

participate in the Tor network as an Onion Router need to run on his computer the Tor software ass relay node or server node. Each Onion Router maintains two kinds of keys. An identity key and an Onion key. The identity key is a long-term key used for signing the Onion Router's descriptor. The Onion Router uses the Onion key, which is a short-term key, for decrypting all the requests from users when they want to set up a circuit. In order to limit the impact of the Onion key compromise, they are periodically and randomly rotated. In order to build the Tor network, each Onion Router maintains a connection to every other Onion Router. That connection is secured by means of the TLS [DA99] protocol. By using the TLS protocol, Tor prevents any attacker from altering the data on the wire or impersonating OR when communicating between OR.

When communicating with each other or with users Ops, Onion routers allow only data packets of fixed size, called cells, to flow along these connections. A Cell is a data structure consisting of a header and a payload. It has a fixed size of 512 bytes. The header has as components ca circuit identifier (circID) identifying the circuit to which belongs the cell, and the command specifying the action to perform on the data. There are two types of cells. Control cells and relay cells. Their distinction is based on the commands. Control cells are made to be interpreted by the receiving node in order to trigger some actions, whereas relay cells are responsible for delivering the stream data.

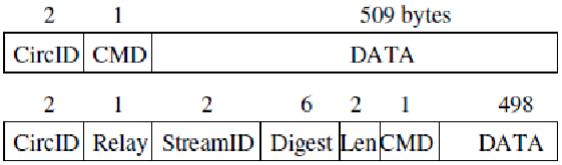

Figure 3.3: Cells header

3.1.4 Directory servers

When the Onion Proxy is about to build a circuit in order to provide an asynchronous connection to the user, how does it know about the existing Onion Routers in the network? Tor uses directory servers for such purpose. A directory server is a well know Onion Router, acting as a

HTPP server, to which Tor assigns the mission of tracking changes in the network topology, node state, key, and exit policies. Each Onion Router regularly provides its current state to each Directory server by means of a signed statement. An Onion Proxy, when needed, can fetch the router list and the current state of the network from the Directory server. In the current system, it does download a Directory list every 15 minutes.

When a new OR joins the Tor network, its publish its state information along with its keys. It does so by sending a signal statement to each Directory server. Existing OR in the Tor network periodically performs the same operation. Upon receiving the signed statement from an OR, the Directory server checks the key of the OR. This operation is necessary to prevent a malicious user from creating many collaborating servers [DD02], which can result on the network being taken over by him. Directory servers proceed to a vote by majority in order to decide to admit or not a new OR. If the result of the vote was positive, Directory servers update their views of the network state by adding this new information to generate a Directory of the entire network. By means of synchronization they then generate a final directory, describing the entire network, which is a result of a consensus among them. A list of the Directory servers along with their keys is preloaded in each Tor client.

3.2 Tor security

From the security point of view, Tor have been designed with two main objectives in mind. The first is to provide a strongly private real time communication over a public network. The second goal is to provide anonymity between the sender and the receiver. By providing the privacy of the communication, Tor intends to prevent a third party or a eavesdropper from listening to the communication on the public network from guessing that the users Alice and Bob are communicating, and from determining the content of that communication. By providing the anonymity of the communication, Tor intends to prevent the receiver Bob, after receiving a message, from being able to identify the sender Alice [STRL01].

In this section we will give a summary of some common and recent attacks on Tor, and we will show how well the Tor design handles each of these situations, if it does. The list of attacks on

Tor is not exhaustive. The various attacks against Tor that have been examined in a recent study [Sal10] can be categorized as follow. traffic and time analysis based attacks, Attacks based on probabilistic models., entry and exit selection attacks, Autonomous system global level attacks, and protocol vulnerabilities based attacks.

3.2.1 Traffic and time analysis based attacks

Traffic and time analysis based attacks is a set of attacks made of by the following attacks. The low-cost traffic analysis [MD05], the cell counter based attack, a browser-based attacks [ALLP07], a congestion attack using long path [EDG09], and a passive-logging attacks [WALS08]. The low-cost traffic analysis attack involves an attacker that is observing the latency of a target Onion Router after creating a circuit loop on that target. He then performs traffics analysis on the latencies data. This attack does not require the attacker to have a global view of the network, instead a partial view of the network is enough for the attacker to perform this attack.

A cell counter attack [LLY+09] is a traffic analysis attack that reveals the end users involved in a Tor circuit. This attack is based on a active watermarking technique. To perform this attack the attacker is required, in a normal case, to have the control of both the ends of the circuit. This attack is based on the idea of introducing in the traffic a cell counter with a secret signal that could be just a well defined sequence of bits. This cell counter, after being injected in the traffic, traverses diverse nodes and then is recognized by the end malicious node, revealing all the parties involved in the communication. There is a variation of the attack performed by a attacker that has only a control of on end.

Unlike the two precedent attacks that are based on traffic analysis techniques, browser-based attacks [ALLP07] are based on timing analysis techniques. The attacker is required to have the control of both the entry Onion Router and the exit Onion Router in order to perform the attack. The two ends Onion Routers could be each in a different circuit. In this attack, the attacker uses the entry Onion Router to perform an analysis of the user's traffic to detect time patterns. He then uses the exit Onion Router to inject HTML or JavaScript based code by modifying the HTML traffic. The injected code can trigger calls to a server that is also malicious. The calls are generated

following time patterns recognizable by the attacker. End to end timing attacks are one of the threats to Tor protocol. Tor has included in this design Entry guards in order to prevent attacks requiring a malicious entry Onion Router. Entry guards are trusted Routers selected following a special algorithm, that are the only allowed to act as Entry Onion Routers. The use of HTTPS protocol and the disabling of active content system, such as JavaScript, are other defenses used against browser-based attacks.

A congestion attack using long path [EDG09] is an attack performed on Tor network and that reveals a user's entire path. This attack is a combination of a selective Onion Router congestion attack and a modification of exit Onion Router's HTTP-stream. The changes in latencies are used to detect the entire path. A successful attack can be achieved when the tree following assumption are met. The first assumption claims that no Onion Router should add artificial delays during packet routing. The second assumption requires the directory servers to provide the list of all Onion Routers. The last stipulates that users have the ability to establish circuits of arbitrary lengths. Tree phases constitute this attack. The first phase is the modification of HTTP-stream on the exit Onion Router. The second is the observation of latencies. The last phase is the performing of the congestion attack. For this attack to be successful the attacker is required to have the control of the exit Onion Router on a victim's circuit. Then the attacker injects either HTML-based code or JavaScript that calls the attacker's web server. To facilitate the monitoring of the latencies, these requests contain a local system time. They are sent using predefined time intervals. The attacker's web server once it receives the requests, stores them and proceeds to their analysis.

Two major factors contribute to the success of this attack. The first one is the fact that most third party software, such as Privoxy and Tor Button, do not disable JavaScript by default, making JavaScript-based modification more transparent to users. The second factor is that a flaw in Tor's path building design, allowing users to build circuits of arbitrary length, facilitates the performing of a congestion attack. Tor design allows a Onion Router only to know its predecessor and its successor. Thus it is possible to construct a long circuit, by imputing the same router repeatedly. An attacker willing to exploit this principle could proceed by first introducing a target Onion Router in the circuit, and then by connecting to at least two other Onion Routers. The long circuit is obtained

by introducing the target Onion Router again and again. This congestion attack is also considered as an effective Denial-Of-Service since it requires only a limited bandwidth from the attackers.

Although Tor's design seems to facilitate the congestion attack, it's a challenging attack to perform. On reason is that searching the routers that will be used during the congestion attack is difficult because all Onion Routers are suspected to belong to the victim's circuit. The other reason is that the attacker's control of the exit Onion Router is limited. Indeed, Tor switches circuit every ten minutes by default.

Two things need to be done to prevent the occurrence of this congestion attack using long path. The one is the disabling of active content systems to prevent the injection of the code modifying a HTTP-stream. The other is to prevent the making of the long path. For this later, a solution would be to limit the path length. But this solution could not prevent an attacker from building multiple circuits that he could then link to obtain a long path. This problem is a challenging one and does not yet have a satisfying solution.

An other subset of traffic and time analysis based attacks is the one made of passive-logging attacks [WALS08]. These attacks involve a predecessor attack and an intersection attack. In the predecessor attack, the attacker keeps the logs of all communications that he suspects them coming from the target user. This is made possible by the fact that the closing and re-establishment of circuits in Tor are frequent, thus making the user's address to be seen more often than others. This allow the attacker to identify the stream originator over time. In the intersection attack, the attacker's logs addresses that have been active during the victim's communication with its destination. Due to fluctuation of circuits in the Tor network, the address list size will decrease, which will increase the probability for the attacker to identify the victim.

3.2.2 Entry and exit Onion Router selection attacks

This category of attacks contain the following attacks. The packet spinning attacks [PASM08] and the low-resource routing attacks. The Packet spinning attack is an attack that compromises the anonymity by using looping circuits and malicious Onion Routers. By using the looping circuit technique, the attacker is able to bloc other Onion Router from being selected in a circuit. Two

assumptions need to be fulfilled to make the attack successful. The first one claims that circular circuits cannot be detected. The second claims that the execution of cryptographic calculation by the legitimate Onion Router will take time. Basically the idea is to create a Denial-of-service attack to target routers by allowing the malicious Onion Proxy to create loops in circuits to these target routers. If the attacker succeed to implement the looping phase, all the legitimate Onion Routers will suffer of a Denial-of-service, except the malicious Onion Routers. This will make the malicious ones to be more likely to be selected in circuits. The attacker can use this advantage to build other attacks.

In a low-resource routing attack [BMG+07], the attacker transmits false resources information to directories. The false information could be a high bandwidth and uptime. These false information can make the attacker to attract an unfair fraction of requests for new circuits. An Onion Router with such a high rate of requests is more likely to be chosen as an entry or an exit Onion Router. After the attacker has increased the probability of his malicious routers to be selected on both ends on the circuit, he proceed to the step which is to expose the circuit by exploiting the victim's circuit creation process. To expose the circuit and identifying the path, the attacker performs an analysis of the patterns in the Tor's circuit building algorithm. While tested in a simulated environment comprising 66 Onion Routers and six malicious Onion Routers, this attack revealed over 46% of paths.

To defend Tor against the low-resource Routing attack, one measure could be to detect faking Onion Routers by verifying the advertised Onion Router data. Another measure could be to require a circuit to be only made of Onion Routers located from different IP addresses and to limit the number of routers per IP address at three. This later measure will increase the resource needed to perform the attack, thus making it costly to the attacker.

Chapter 4

Basics in Cryptography

4.1 Introduction

We would like to make this paper to be self contained and uses cryptographic terminology that is already defined. Hence, we will devote this chapter to the introduction of cryptographic notions and terminology that will be used throughout this paper. This introduction does not pretend to cover all aspects of cryptography. Its purpose is to help readers without prior cryptographic knowledge to be able to appreciate this work.

The composition of this chapter is as follow. First we introduce a brief history of cryptography and then we define the terminology used in this work, followed by a description of the important attacks on cryptographic systems. This will bring us to a discussion about the two sets of encryption schemes that are the public key cryptography and the secret key cryptography. That discussion will close this chapter.

4.2 Brief history of cryptography

The history of cryptography is made of the development of techniques enabling the hiding of the information. We can divide the time-line of the development of cryptography as follow. The beginning, the transposition cipher era, the rotor machines era, and the digital cryptography era.

4.2.1 The beginning

The oldest known text to contain one form of cryptography is believed to be the hieroglyphic inscription on the tomb of KHNUMHOTEP II, an Egyptian nobleman of the town of MENET KHUFU. These inscriptions where written with unusual symbols in order to obliterate their meaning [Sin00].

4.2.2 The Transposition era

In a transposition cipher, the sender disturb the order of characters by shuffling them around, whereas the plain text remains the same. One of earlier use of the transposition cipher is believed to have occurred in SPART, a Greek city, in % BC. The Spartans are believed to have developed the SCYTALE, a cryptographic device allowing to send and receive secret messages. The device was a cylinder that was performing a transposition of characters, and was possessed by both the sender and the receiver [Sin00]. Using the SCYTALE, the secret message was obtained as follow. The sender wound a narrow trip of parchment, similar to modern day paper, around the SCYTALE. He then write his message across that parchment. Once done, the tape is unwounded and presents a sequence of meaningless characters. That's the secret message. To recover the original message, the receiver needs to have a SCYTALE of exactly the same diameter as the one used by the sender. He will rewound the tape onto his SCYTALE in order to get its meaning [Sin00].

Figure 4.1: The Scytale

More recently, during World War I, the German developed a device called SDFGVX that was performing a transposition cipher combined with a simple substitution. The device produced an

algorithm that was very complex at that time, but ended up being broken by a French cryptanalyst [Sin00]. The high requirement of memory and the limitation of the length of the message that can be processed make the transposition cipher less and less used although some modern algorithms still have it as a component.

4.2.3 The Rotor machines

The era of Rotor machines saw the introduction of mechanical devices able to automate the process of encryption. A rotor machine is constitute by a keyboard and a series of rotors. Each rotor performs a simple substitution of characters. The rotor is designed to represent an arbitrary permutation of the alphabet, having hence 26 positions. The rotor machine performs an encryption as follow. Let consider a 4-rotor machine. The first rotor, receiving the plain text, might be wired to substitute D for A. The output of the first rotor is wired as input to the second rotor. The second rotor in turn might substitute F for D, the third rotor might substitute T for F and the last rotor might substitute Z for T. Therefore, by this process, the output for the input character A would be Z [Sin00]. The way several rotors are combined and the fact that they move at different rate contribute to make the all system secure.

Figure 4.2: The Enigma

One of the best representative of rotor machines is the notorious German ENIGMA [Win05]. It was invented by two German scientists Arthur Scherbius and Arvid Gerhard. It was widely used

by the German army during World War II. The Enigma was build with five rotors but each time processing three rotors out of the five. It has also a pluggable keyboard that performs a permutation on the plain text as a pre-processing. The breaking of the Enigma, first by the Pols, was a major accomplishment that brought a major contribution to the victory of the Allied forces during World War II.

4.2.4 The Digital Era

The advent of the microprocessor and the personal computer have brought our every-day life into a digital world. This has contributed to a build up of vast communication network such as Internet, Mobile phones. The success of these Network and the huge amount of communications they drain have motivated the need, among Government or among citizens, of new kinds of cryptographic tools that pertains to the digital message. The trend starts in the early 1970's when IBM offered to the US Government an algorithm based on the IBM LUCIFE algorithm. This was the beginning of the Data Encryption Standard or DES. DES is a 64 bits symmetric block cipher.

This early symmetric encryption system suffered from the problem of the key exchange between the sender and the receiver. In 1975 WHITFIELD DIFFIE and MARTIN HELLMAN [DH76] developed the concept of ASYMETRIC KEY which opened the possibility for the sender and the receiver to use different keys (The public and the private), eliminating the problem of keys exchange inherent to symmetric encryption.

There are much to say about the history of the cryptography. This is not intended to be an exhaustive treatment of the subject. A book that presents a good overview of the subject is [Sin00].

4.3 Terminology

In this section we are going to provide some definition of basic cryptographic terms and notions used throughout this paper.

Bruce Schneir [Sch95] defines cryptography as The art and science of keeping messages secure. The word cryptography derives from two Greek words that are Kriptos, meaning Hidden, and Graphos, meaning Writing [AJP95]. Cryptography can be assigned many goals, but three of them

are considered as the principal ones. The first goal is authentication. Authentication provides to the computer application the assurance that users are who they say they are. The second goal is confidentiality. Confidentiality provides to the users a way to protect their data from unauthorized access. The last of the three goals is integrity. Integrity is ensuring users that their data have not been modified by unauthorized third party.

Before the modern times, cryptography was almost exclusively involving the encryption. Encryption is refereed to as the process of converting an original and readable message, that is called a plaintext, into an unintelligible message, that is called a ciphertext [Kah96]. the reverse of the encryption process is called a decryption. In other words, it's a process of transforming the unintelligible message (the ciphertext) back to a readable message (the plaintext). A cipher is a pair of algorithms whose one is responsible to perform the encryption and the other one is responsible of the decryption. Each cipher needs the combination of both the algorithms and a key. A key is a secret parameter that should be known only by the actors involved in the communication. The ordered list of elements of finite possible Plaintexts, finite possible cyphertexts, finite possible keys, the encryption and decryption algorithms corresponding to each key, form what we call a cryptosystem [Kah96]. Cryptology, which refers to the study of cryptosystems, is often divided into two disciplines, that are cryptography and cryptanalysis [Gol01]. Cryptography refers to the study of the design of cryptosystems, while cryptanalysis concerns itself with the breaking if cryptosystems. These two aspects of cryptology are considered to be related; when designing a cryptosystem one may need to use cryptanalysis in order to analyze the strength of its security[Til02].

4.4 Attacks

We can distinguish a wide variety of cryptographic attacks that one can classify in any of several ways. The principal parameters to distinguish those attacks are as follow. What an attacker knows and what capabilities are available [AJP95].

Basically, we often distinguish two types of attacks. Active attacks and passive attacks. Active attacks are the ones in which the attacker cause a modification, a delay, a reordering, a duplication, or a synthesis on the data. These attacks offer three kind of threats on the observed data:

- The authenticity attack that creates a doubt of the origin of the data.
- The integrity attack that modify the content of the data.
- The ordering attack that modify the ordering of the data bites arriving at destination.

The passive attacks involve the following attacks. In a ciphertext-only attack, the cryptanalyst is required to capture a segment of the ciphertext. He will attempt to determine the corresponding plaintext, and if possible, the key that was used with no other information than the segment of the ciphertext previously caught. This attack is considered as the most difficult form of attack against a cryptographic system. In a know-plaintext attack, the intruder is required to capture, in some manner, a segment of ciphertext and the corresponding plaintext. Using these two pieces of information, he will try to discover what key was used. The final purpose is to try to decrypt other information that have been encrypted using the same key. In a chosen-plaintext attack, the cryptanalyst is required to chose a plaintext, and can somehow learn the ciphertext associated with the plaintext based on some special characteristics or specific patterns of the plaintext. This attack is considered to be the most dangerous attack. Finally, in a chosen-ciphertext attack, the intruder may capture a ciphertext and then try to learn the corresponding plaintext [AJP95].

4.5 Symmetric cryptography or Private-key cryptography

The private key cryptography, also called symmetric cryptography, have been defined as follow [Pie00]. Let consider M as a set of all possible plaintext messages, C as a set of possible ciphertext messages, and K a set of all possible keys. We consider a private key cryptography as a family of pairs of functions

$$e_k : M \rightarrow C, d_k : C \rightarrow M, k \in K \tag{4.1}$$

such that

$$d_k(e_k(m)) = m \text{ for all } m \in M \text{ and all } k \in K \tag{4.2}$$

. It's a cryptosystem that uses the same key for both the encryption and the decryption of the messages.

The symmetric key cryptosystem has two requirements[MOV97]:

- 1. The first requirement involves the assumption that it is impractical to decrypt a message while having the knowledge of the ciphertext and thee one of the encryption / decryption algorithm;
- 2. The second requirement stipulates that the sender and the receiver must arrange to share the key somehow and they must ensure the secrecy of that key. The knowledge of both the key and the algorithm by an attacker can lead him to read all the messages using that key.

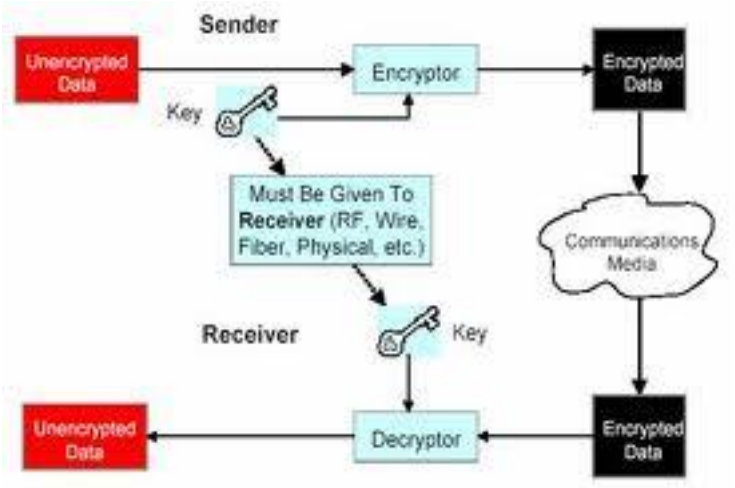

Figure 4.3: Symmetric encryption diagram

Private key cryptography are of two types. The ones using stream ciphers and those one using block ciphers. Stream ciphers operate by encrypting the bits of a message one at a time. Whereas block ciphers proceed by taking a block of bits and encrypt it as a single unit. Well established and popular symmetric algorithms include DES, Triple DES, Twofish, Serpent, Blowfish, CAST5, RC4, IDEA, and AES [MOV97].

Besides doing the encryption, private key cryptosystems are also used to a mean to provide message integrity. Since encrypting a message does not ensure about its integrity, a message authentication code is produced from the ciphertext and provided, along with the ciphertext, to the receiver as a proof of the authenticity of the message. The receiver can use it to verify that the message have not been modified before being received.

Private-key cryptosystem have been used in a broad range of applications. However, it is unsuitable for use in certain applications because of the following disadvantages [Pie00]:

- Key distribution problem : the two users have to select an available secure channel over which they can exchange the chosen key. This secure channel may not be available.
- Key management problem : Since the key should be known only by both the sender and the receiver, every pair of users must share a key. For a network if n users, the total of n(n-1)/2 keys is needed. If n become large, then the number of keys can become difficult to manage.
- No signature possible: In this cryptosystem, the sender and the receiver are using the same key for encryption and decryption. Therefore the receiver cannot convince a third party that the message he received from the sender truly was emitted by the sender.

4.6 Asymmetric cryptography or Public key cryptography

Public-key cryptography refers to a cryptographic system that uses a pair of separate keys, one called public key and used for encryption, and the other one called private key and is used for decryption. Neither key is able to perform both functions. These public and private key are mathematically related to each other.

Diffie and Hellman were the first to show publicly the feasibility of a public-key cryptography. This was done during the presentation of the Diffie-Hellman Key exchange protocol [DH76]. In that paper published in 1976, Whitfield Diffie and Martin Hellman introduced to the world the notion of public-key cryptography that uses a pair of separate but mathematically related keys, one called a private key and an other one called a public-key. 1978 marks a milestone in the history of the public-key cryptography by the introduction of a new public key system called RSA, by Ronald Rivest, Adi Shamir, and Len Adleman [RSA78].

4.6.1 How it works

Let consider two users, Alice and Bob, who are willing to exchange a secret message. To use asymmetric cryptography, the process begin with Bob randomly generating a pair of Public/private

keys. Bob then publish his public key ti allow everyone to access to it, Alice included. Then, when Alice decides to send some secret message to Bob, she will select an appropriate asymmetric algorithm and uses it with both the public key received from Bob to encrypt the message. The encrypted message will be sent to Bob by Alice. Upon receiving the ciphertext from Alice, Bob will decrypt it with its private key. Since this private key matches the public key used to encrypt the message, the decryption will be done easily by Bob. Anyone else who does not have the private key matching the public key used to encrypt the message will have a hard time to decrypt it.

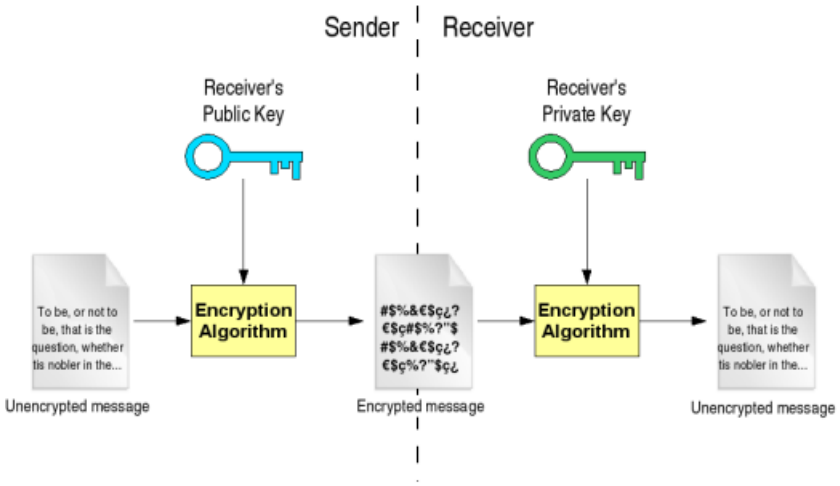

Figure 4.4: Asymmetric encryption diagram

This schemes presents many advantages in many situations, especially if the message exchange involves two parties that have no previous contact with one another. However, one criteria for this scheme to fully work, is the ability for Alice ti have the guaranty that the public key she is using came truly from Bob. This problem is known as the public key authentication problem. It is solved by introducing a third party into the scheme, whose job is to provide the authenticity of the public key [Ray10].

The private key is always kept secret by the party that generates it. It does not need to be communicated to any other party. Therefore, in many situations the public/private keys pair may be kept unchanged and used for a long period of time. This makes the total number of required keys in a public-key scheme much smaller than in a private-key scheme, even for a large network [MOV97].

4.6.2 Properties of public/private keys

For a public/private keys pair to be suitable to be used in a asymmetric cryptosystem, it should fulfill some properties like the following:

- The public/private keys pair should be computationally easy to generate by Bob.
- It should be computationally easy for Alice to generate the ciphertext if she knows Bob's public key and the plaintext.
- Decrypting the resulting ciphertext should be computationally easy for Bob if he uses his private key to produce the original plaintext.
- Although the public and private keys are mathematically related, it should be computationally infeasible for a third party (Charles) to recover the original plaintext or to reveal Bob's private key if he knows only Bob's public key.

By computationally easy we mean computing a function in a polynomial time, since in a public cryptography both encryption and decryption involve computing functions. The polynomial time is expressed as a function of input length [Sta11].

4.6.3 Using public-key cryptography

Since the computational performance of public-key encryption is inferior to that if symmetric-key encryption, it is not often used for encryption/decryption purpose. However, when dealing with a small load of data, one may use an asymmetric cryptosystem to perform the encryption and decryption. This is called ensuring the confidentiality of a small load [Ray10]. Another case of using an asymmetric cryptosystem for encryption and decryption purpose is to solve the symmetric key exchange problem [MOV97]. In this scenario, if Alice is willing to share a secret with Bob, she will look up Bob public key and hen used it to encrypt her public key along with her part of a session key. Then she sends both keys to Bob. Upon reception, Bob will decrypt the message with his private key. He then combines Alice key part with his own part. The resulting key is encrypted with Alice public key and sent back to Alice. When Alice receive the response from Bob, she

decrypts it with her own private key and then recover the whole key session. This is also called ensuring the confidentiality of the session [Ray10]. In addition, a public-key cryptosystem can be used to perform digital signature when the private key is used to sign a message and the public key is used to verify the signature. This is how an integrity of a message can be guaranteed, and the non revocation of a key can be achieved.

Chapter 5

The Design

5.1 Introduction

The architecture we are implementing is made of four different protocols, the protocol for anonymous upload Tag generation, the protocol for anonymous upload and retrieval, the protocol for anonymous delete. We assume that the anonymous communication is provided by an anonymizing network, for instance tor network. We assume that two clients named Alice and Bob are willing to exchange files through the supporting Server named SupServer. Both Alice and Bob at some point will generate public and private keys for cryptographic purpose. We assume that both Alice and Bob public's keys are publicly known, included to the SupServer. In this chapter we describe how each protocol is designed.

5.2 Protocol for anonymous upload tag generation

We want Alice and Bob to be able to upload anonymously files into the SupServer. This will prevent the SupServer from linking each file it is storing to its owner. This will also prevent any other party from linking files into the SupServer to their owners. The issue with this design is how does a subscriber that wants a file that belongs to a particular publisher? The solution to this problem is to append a key value to the file, like an index, that will allow the file to be uniquely identified into the SupServer while still hiding the identity of its owner. This key value, as a unique value, is also used by the subscriber to retrieve a particular file. For this to happen, both the publisher and the subscriber need to share the knowledge of the same key value. This raises

the challenge of making two entities to share the same secret through the server, without letting that server to know the secret. The upload tag is a unique value that a publisher appends to a file, like an index, to help uniquely identifies this file. It is generated by the publisher in collaboration with the subscriber, so that both can share the same secret. This section explains how this protocol proceeds to generate an upload tag.

5.2.1 The upload tag

To solve this issue, each Alice's file stored on the SupServer will be indexed by a Tag that was anonymously generated by Alice in collaboration with Bob. Prior to exchange any file through the SupServer, Alice and Bob will generate the upload tag by using available public information. The Tag has this interesting property that it cannot be linked either to Alice, either to Bob. Moreover, knowing the Tag may give a third party the capability to retrieve a file from the SupServer, but the lack of appropriate keys will prevent him from reading the file. Therefore, it is not necessary to encrypt the tag. Let now see how Alice and Bob manage to generate the same Tag.

5.2.2 Tag generation

In this section we describe how Alice and Bob manage to generate the same Tag. For this purpose, we will use the symbols described in Table 5.1 [Ray09].

A Tag is actually a hashed Diffie-Hellman secret shared between Alice and Bob. To generate that Tag, they proceed as follow:

1. The protocol starts with Alice and Bob making agreement on α and P with α un element generated by a given cyclic group generator G, and P a large prime number such that $P >> \alpha$ and α is a primitive root of P. This agreement can be made in a offline manner. The values α and P can be known to an adversary without creating any harm for the security of the Tag.

2. The next step of the protocol sees Alice and Bob generating respectively a secret random number r_A, and a secret random number r_B. Alice then compute the value $\alpha^{r_A} \bmod p$, which is the half of the shared secret. She then get T_A, with $T_A = Sig_A[\alpha^{r_A} \bmod p]$. The next steps sequentially involve encrypting T_A with a secret key k_1, encrypting the

Table 5.1: Symbols used in protocol discussions

Symbol Used	Interpretation
$E_x^{pub}[M]$	Encryption of message M with public key of entity X
$E_k[M]$	Encryption of message M with secret key k
$M_1 \mid\mid M_2$	Concatenation of messages M_1 and M_2
$H_k[M]$	Cryptographic hash of message M with key k using a hard to invert hash function
$rand_k(\cdot)$	A cryptographically strong pseudo-random number generator using a secret key k that is unique to each entity
$ID(X)$	Publicly known identifier of entity X
TS	A global timestamp
N	The null value
$Sig_x[M]$	Message M signed with private key of entity X
$H[M]$	Message digest of message M generated using a hard to invert hash function
$H^n[M]$	Message M hashed n number of times using hash function H

secret key k_1 with Bob public-key, generating a random number, and generating a timestamps. By putting all these pieces together, Alice will obtain the following message : $ID(A), TS, N, E_B^{pub}[k_1], E_{k_1}[T_A] \mid\mid E_{k_1}[m]$. This message is considered as a half tag object, and will be uploaded on the SupServer. Similarly Bob also starts by computing the value r_B followed by $\alpha^{r_B} \bmod p$, which value he sign to get $T_B = Sig_B[\alpha^{r_B} \bmod p]$. He goes also through the same following steps as Alice to get the message $ID(B), TS, N, E_A^{pub}[k_1], E_{k_1}[T_B] \mid\mid E_{k_1}[m]$ that he will upload on the SupServer. These messages can hold Alice and Bob information without presenting any security risk for the Tag generating protocol.

3. Since the communication between Alice and Bob is asynchronous, there will be a time when Alice and Bob will download each the other half Tag message. After performing the necessary decryption, Alice and Bob will use the remaining value from the downloaded half

Tag message to independently compute the Diffie-Hellman secret key $\alpha^{r_A r_B} \bmod p$. To get the upload Tag, each of them independently hashes its Diffie-Hellman secret key. The hash function has the property of being hard to inverse [DH76]. Hence, it guaranteed that nobody can recover the Diffie-Hellman secret key by inverting the hash function. The following figurexxx gives a clear view of all the process of generating the upload Tag [Ray09].

5.3 Protocol for Anonymous upload and retrieval of message

All clients communicate anonymously with the SupServer. This later cannot link files to their owners. For this reason, a Tag is used as a file index to help any client to identify a file on the SupServer. Let see how Alice and Bob do to upload and retrieve files from the SupServer.

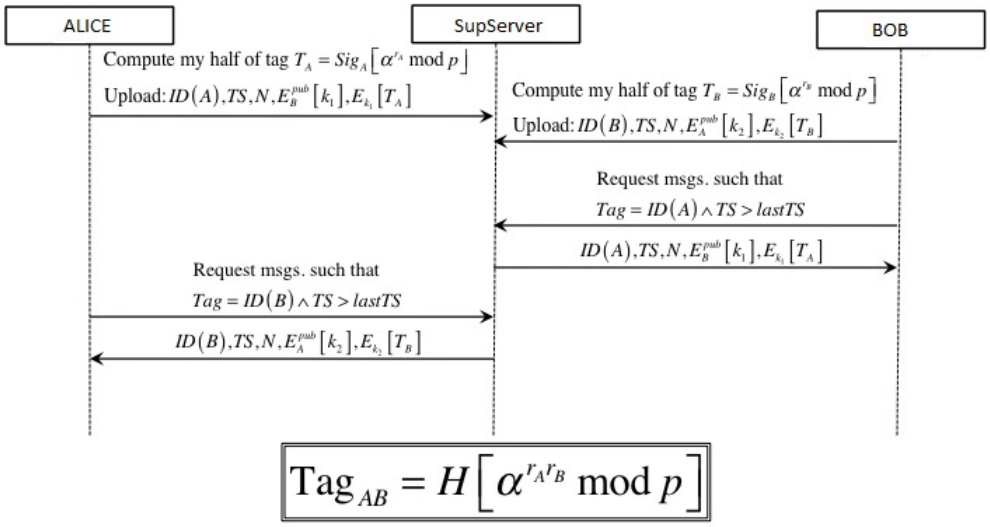

Figure 5.1: Upload Tag generation

5.3.1 The upload protocol

Before Alice upload any data to the SupServer, she will create two Tags, the upload Tag, and the DeleteTag. We have seen how to generate the upload tag in the previous section. The DeleteTag is an unique value that Alice will generate as following: DeleteTag $= H^n[rand_k(\cdot)]$. To avoid the DeleteTag to fall on bad hands, it is encrypted with the SupServer public key to get the value : $DT = E^{pub}_{SupServer}$[DeleteTag].

After computing these two Tags, Alice will make a formatted message that she will upload to the SupServer. This formatted message is structured as shown in the figure 5.2:

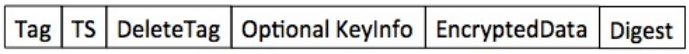

Figure 5.2: Message format

Where Tag is the upload Tag generated by Alice in cooperation with Bob, TS is a timestamps, DT is the DeleteTag encrypted with the SupServer public key, $E^{pub}_A[k_1]$ is Alice secret key encrypted with Bob public key, EncryptData is the data encrypted with Alice secret key.

5.3.2 The retrieval Protocol

The SupServer stores data as formatted message as shown in the previous section. Anyone who needs to retrieve a message from the SupServer needs the appropriate Tag. Since Bob cooperated with Alice to create the upload tag, Bob can use it to retrieve Alice's file from the SupServer.

To avoid Bob from retrieving several of Alice's files, Bob query can be refined by including some parameters pertaining to the specific file, and comprised in the formatted message. The refined query will be expressed as following: Tag $= Upload - tag \wedge$ TS $> lastTS$.

The retrieval protocol process can be seen on the following figure 5.2.

5.4 Anonymous Deletion

Any client needs to prove that he has the privilege to delete a file before any deletion request is executed by the SupServer. Since Alice uploaded her file anonymously on the SupServer, the

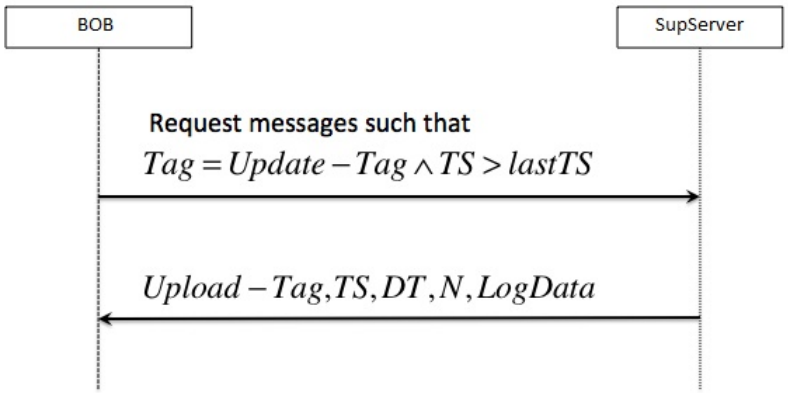

Figure 5.3: Protocol for data retrieval

SupServer does not have a way to verify the file's ownership when someone requests to delete the file. To address this problem, the SupServer will have to ask anyone requesting the deletion of a file to prove that he has that privilege. The DeleteTag is used for such purpose. In addition to the DeleteTag, the upload Tag will be needed as a proof of the cooperation with the file's owner.

To delete Alice's file, Bob proceeds as follow:

1. Bob will send to the SupServer a request for a message fulfilling the following condition: Tag $= Upload - tag \wedge$ DeleteTag $= DeleteTag$

2. The SupServer will request Bob to prove that he has the privilege to delete the requested Alice's file.

3. Bob will prove that he has the privilege to delete the specified Alice's file by presenting the value $H^{n-1}[rand_k(\cdot)]$.

4. The SupServer uses the proof presented by Bob to verifiy that $DeleteTag = H[H^{n-1}[rand_k(\cdot)]]$

If the result is equal to the DeleteTag, then Bob's delete request is granted, and the SupServer executes the delete operation. Otherwise the delete request will be rejected by the SupServer. This process can be seen on the figure 5.4.

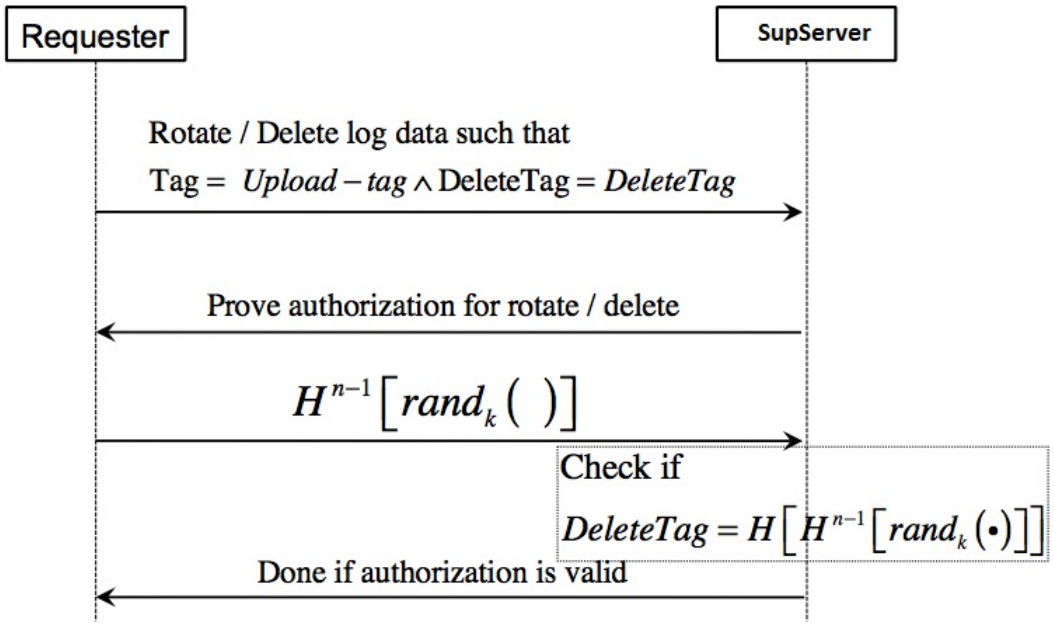

Figure 5.4: Protocol for data deletion

5.5 Analysis

This protocol was developed with the main objective of allowing all clients exchanging messages through the SupServer to stay anonymous in regard of the SupServer, while still able to perform all operations. With the assumption that the SupServer provider is honest but curious, let see how this anonymity is guaranteed.

Although the SupServer is curious, it does not have enough information to identify the clients exchanging messages through it. First of all, all messages being exchanged are done so over the Tor network. The Tor network masks the identity of the clients logging ti the SupServer. In addition, the SupServer cannot link any message it received to its owner. The first messages exchanged are the public parameters or public key, and the half Tag. Although these data contains identify of their owner, the SupServer cannot link them to their generator because of the use of Tor network.

The other data being exchanged are files to which are appended upload Tags. Here again, the SupServer cannot use the tag to identify its generator. The Tag being the result of a hash function over a Diffie-Hellman secret key, it is hard to inverse it in order to recover the secret key. Even

though it succeeds to recover the secret key, b the property of the Diffie-Hellman algorithm, it cannot establish the link between the public parameters and the secret key.

An attacker that succeeds to break the SupServer and to get its data will have the same limitation toward these data as for the SupServer. It still cannot link the data to its generator. If the attacker decide to intercept the data during its transmission over the Tor network, on one hand it cannot link that data to it originator because of the Tor network, and on the other hand it cannot replay that data to the SupServer because the data is sent as a formatted message with a field filled with the timestamps.

Chapter 6

The Challenges

6.1 Introduction

One of interesting aspect of our work was to make all communications to be anonymous. In order to reach that anonymity we have decided to make use of an anonymizing software. There are many products out there claiming to be able to provide communication anonymity. A survey made in [CC05] shows that Tor is the most efficient among all anonymizing programs. Therefore we have made the design decision of using the Tor network to anonymize our communication. In this section we highlight some of the challenges we faced when trying to anonymize our communication through the Tor network.

6.2 Problematic of using Tor

Using Tor raises the issue of enabling your application to connect to the Tor network. In order to use Tor, one needs first to install a Tor client on his local host. The Tor client is responsible for enabling you the access to the Tor network. In addition one needs to make his application able to communicate with the Tor client. Since this later uses the SOCKS protocol, the issue is how to make your application SOCKS aware in order to communicate with the Tor client.

6.3 SOCKS protocol

SOCKS [Lee] [WIK12] stand for SOCKet Secure. It is an Internet protocol that allows a client and server to communicate by routing the client packets through a proxy server. The last version

provides also authentication so that a server can be accessed only by authorized users. SOCKS is a protocol designed at the session layer of the OSI/model [WIK12].

6.3.1 Usage

The SOCKS protocol can have several uses. An user can free himself from the limitation of connectivity only to a predefined remote port and server by creating a local SOCKS proxy by mean of use of some SSH clients that supports dynamic port forwarding. We already mention that the Tor client, which acts as a proxy, presents a SOCKS interface to its clients. Another interesting use of SOCKS is as a tool allowing users to bypass government or any other organization Internet filtering system in order to access blocked contents [WIK12].

6.3.2 Interaction with the firewall

Many organizations set Internet filtering rules by means of a Firewall to prevent their users from accessing any other Internet service besides the web service. The SOCKS protocol can be used to bypass those firewalls. To do so, the protocol SOCKS will be used as a tool to create a RAW TCP connection to the proxy server. Then that proxy server is used to communicate the IP address and the port to which the connection is requested. The proxy server can decide, at its own discretion, whether to grant, reject or redirect the connection from the requesting client [WIK12].

The SOCKS proxy can be used as follow. Alice and Bob are willing to communicate in a network where a firewall stands between them, and prevent any communication between them. Alice is the initiator of the connection. She will initiate a connection to the SOCKS proxy and transmit to it the host and port she want to use to connect to Bob. The SOCKS proxy will bypass the firewall, and will make a connection to Bob on behalf of Alice. Bob response will follow the same process.

6.3.3 Protocol

The SOCKS protocol was originally created as a tool to facilitate firewall and other security tasks administration. It can be found now under three versions: SOCKS4 [Lee], SOCKS4a, and

SOCKS5 [LGL+96]. SOCKS4 is the basic version of the protocol. It does not allow users authentication, and allow only TCP connection in addition of not supporting Ipv6 addresses. SOCKS4a is an extension of the basic version. It adds to the basic version the possibility for users to perform a domain name resolution. SOCKS5 is the last version as of today, and has more features than the two previous versions. It supports Ipv6 addresses, and allows UDP traffics besides allowing users authentication.

6.3.4 Software

The SOCKS protocol is implemented as SOCKS server and SOCKS client. The following products can be considered as SOCKS servers: Dante, OpenSSH, PuTTy, Wingate, SS5, and Tor. In order to connect through a SOCKS proxy, client applications should support the SOCKS protocol. If they don't, there are programs, the SOCKS clients, that can allow them to overcome that limitation. These SOCKS clients are also called proxifiers. The following programs can be considered as SOCKS clients : Socat, Polipo, Dante client, Freecap, tsock, Privoxy, Proxifier, ProxyCap, Proxychains. This list is not exhaustive, a more detailed description can be found at [SER12].

6.4 Connecting to Tor

Finding the right program allowing our client/Server applications to connect to Tor revealed to be a challenging task.

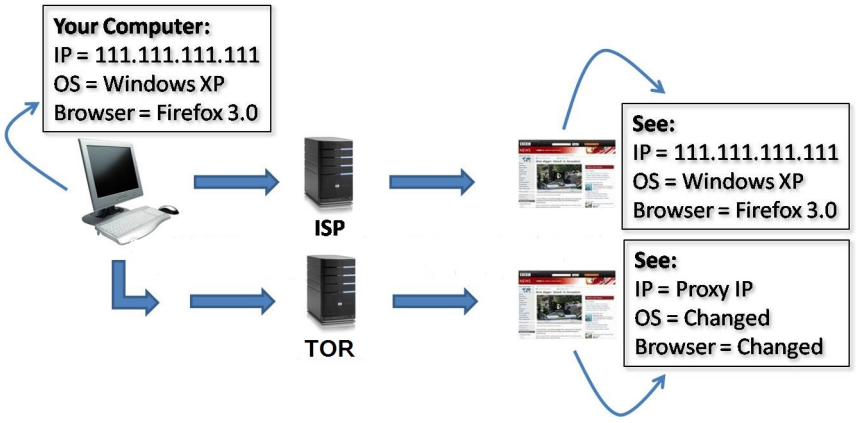

Figure 6.1: Why connecting to Tor

Since our client/server applications are not SOCKS aware, we need to use a third party program to allow them to communicate through Tor as shown in figure 6.1. In order to reach that objective, we have successively tried the following programs : Polipo, Privoxy, Tsocks, Dante, and Socat.

6.4.1 Polipo

Our first introduction to the matter of connecting our client application to the Tor network was with Polipo. Polipo [Chr08] is a lightweight and fast proxy server that can be used as a web cache, an HTTP proxy, or a proxy server. It is a free software that run on various blend of Linux, on Windows, and on MacOS. It is a small proxy. But it is among the proxy with the richest features. Polipo provides the following features: It supports HTTP 1.1 and can speak both IPv4 and Ipv6. This feature allow Polipo to be used as a bridge between Ipv4 and Ipv6. Polipo can serve as a proxy for a single computer or for a small network.

6.4.1.1 Configuring Polipo

Polipo configuration file is responsible for the gathering and the setting of the numerous configuration variables. The configuration file is either /.polipo or /etc/polipo/config, depending of which one exists. Polipo can be customized to your liking by tweaking its numerous configuration variables. The following command can list all the variables.

$$\$polipo \quad -v$$

A detailed description of all the variables can be found in [Chr08].

6.4.1.2 Using Tor with Polipo

An application that is not SOCKS aware can use Polipo in order to connect to the Tor network. Polipo, by its numerous features, is even said to make slow network, like Tor, to appear faster. We have performed the experiment of connecting Firefox to Tor through Polipo. To do so, the three components involved in the communication, Firefox, Polipo and the Tor client should be properly configured. To configure Firefox, you open the browser, and you click on the menu Edit Preferences. In the wizard clicks the connection settings button, and select the manual proxy

configuration. You need to fill the following information in the top field. HTTP Proxy : localhost, Port : 8123. These are the port and address to reach Polipo.

The next step is to tell Polipo to use Tor. To do so, you need to edit the following variable in Polipo configuration file.

$$socksParentProxy = 9050$$

The port 9050 is the default port where Tor listen to client's connections. At this step, the three components, Firefox, Polipo, and Tor client are ready to participate in a connection. Unfortunately, Polipo turns out not to be the right candidate for connecting our client application to Tor. The reason was that it is a web proxy and our client application is not a web one. Before moving away from Polipo, we have also tried Privoxy. Privoxy is used for the same purpose as Polipo, and it behaves exactly like Polipo. We don't found it relevant to describe it in more detail. Therefore we have to try another program that is not a web proxy.

6.4.2 Tsocks

After failing to use Polipo in order to connect our client application to Tor, we move to another choice of program. We need a program that is not specifically only a web proxy. The new candidate is Tsocks.

Tsocks [Clo08] is a program intended to allow network applications that are not socks aware to be able to gain access through SOCKS proxy without suffering any modification. Tsocks use the shared library interceptor concept as its basic concept. Tsocks makes use of the $LD\_PRELOAD$ environment variable to get automatically loaded into the process space of a program, whenever a network based application is executed. From there Tsocks intercepts all $connect()$ calls to establish a TCP connection, and replaces them by its own version of $connect()$. It then negotiates the connection after checking the configuration file to see if the connection needs to go via a SOCKS server. If the negotiation succeeds, it passes the connection back to the requesting application. To preload the $LD\_PRELOAD$ library the following commands can be used:

$$export\ LD\_PRELOAD = libtsocks.so \text{ or } setenv\ LD\_PRELOAD = libtsocks.so$$

6.4.2.1 Tsocks configuration

As with Polipo, Tsocks configuration is done through the editing of the configuration file. Most configuration options are provided into that file, whereas some can be specified at run time. The configuration file is by default $/etc/tsocks.conf$, but you can change this location. In that case you need to provide the location of the configuration file at configure time by writing this argument:

$$-with-conf = new\_location\_of\_configuration\_file$$

6.4.2.2 Using Tsocks with Tor

We have tried the experiment of connecting our client application to Tor using Tsocks. Tsocks and tor need to be properly configured in order for the sooner to routes all connections to Tor. The following is an example of Tsocks configuration to allow it to work with Tor.

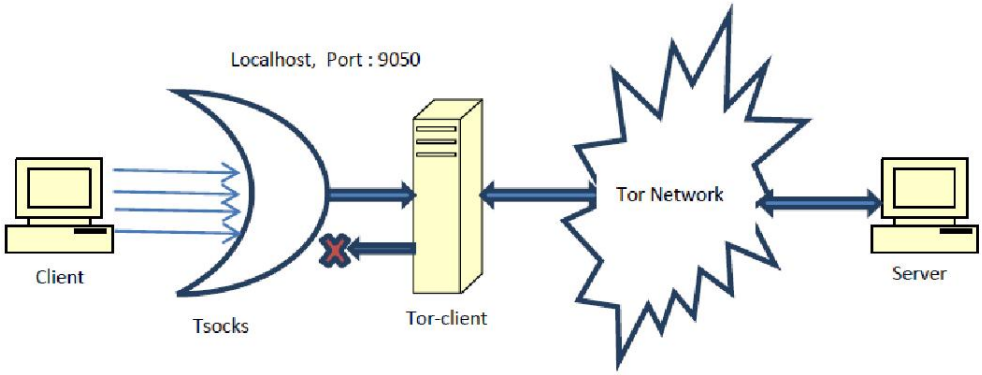

Figure 6.2: Using Tsocks with Tor

After messing with Tsocks and Tor configuration, we were not still able to allow our client application to connect to Tor. The reason happens to be that Tsocks can only process outgoing connections. It lacks the $BIND$ command necessary to process incoming connections. From this experiment, we understood that we needed a program like Tsocks , but that is able to process connection both ways.

6.4.3 Dante

Dante [DA202] is client/server implementation of the SOCKS protocol. It was developed by Inferno Nettverk, and is released under a BSD license. Once installed on a host, Dante can provide a transparent connectivity to clients running on that host while still make it possible for server administrators to have detailed access control and logging facilities. The basic version of Dante can be enriched with several commercial modules provided for such purpose [DA302]. Let give a close look at the two parts, Dante-client and Dante server, that make up Dante.

6.4.3.1 Dante-client

Dante-client is a client side of the program Dante. It provides a SOCKS wrapper for no SOCKS aware applications that are behind a firewall. There several different ways of using the Dante-client library. Either by compiling that library into your client application, or with the $socksify$ command. Transforming an usual no SOCKS aware application into a SOCKS client is referred to as socksifying the application, and is accomplished by using the following command [DA402].

$$socksify\ \ my\_client\_application$$

By calling $socksify$ the $LD\_PRELOAD$ environment variable will be automatically set to the $libdsocks$ library residing in the $libdsocks.so$ file. If your platform allows the setting of $LD\_PRELOAD$ environment variable, $socksify$ will preserve you from recompiling your client application in order to redirect the system networking calls through the proxy server. Another thing to keep in mind about the $LD\_PRELOAD$ is that setuid applications usually ignore the $LD\_PRELOAD$ environment variable [Kak11].

6.4.3.2 Configuring Dante-client

In order for Dante-client to work, some configuration information need to be provided. Two ways can be used to provide these information, either via the shell environment, or via Dante-client configuration file. You cannot control all aspects of the client application using only the environment. In many usage scenarios only the $SOCKS\_SERVER$ variable setting can be done

that way. This variable controls the proxy server address. It can be set by using the following command:

$$export\ \ SOCKS\_SERVER = 192.0.5.6$$
$$socksify\ \ my\_client\_application$$

Dante-client configuration file is /etc/dante.conf, and is more preferred when a greater control of the socksified application is needed. The Dante-client configuration file contains several rules to control the client application, and the most significant rules are the route rules. These rules control the client application communication with other hosts, and are the only required part of the configuration file. Although there is no limit to the number of routes that can be specified in the configuration file, in many usage scenario it is specified a route for outgoing connections to a proxy, a route for incoming connection from the server, and an optional direct route to a DNS server [DA402].

6.4.3.3 An example of Dante-client configuration file [Kak11]

See the actual file $/etc/dante.conf$ in your own installation of Dante for further details.

#debug: 1

Allow for "bind" for a connection initiated by a remote server

in response to a connection by a local client:

$route\{$

$from : 0.0.0.0/0\ \ to : 0.0.0.0/0\ \ via : 127.0.0.1\ \ port = 1080$

$command : bind$

$\}$

Send client requests to the proxy server at the address shown:

$route\{$

$from : 0.0.0.0/0\ \ to : 0.0.0.0/0\ \ via : 127.0.0.1\ \ port = 1080$

$protocol : tcp\ \ udp \qquad \#server\ supports\ tcp\ and\ udp.$

$proxyprotocol : socks\_v4\ \ socks\_v5 \qquad \#server\ supports\ sock\ v4\ and\ v5.$

$method : none \qquad \#username \; \#we \; are \; willing \; to \; authenticate \; via$

$\#method \; "none", \; not \; "username".$

}

Same as above except that the remote services may now be named

by symbolic hostnames:

$route\{$

$from : 0.0.0.0/0 \;\; to : . \;\; via : 127.0.0.1 \;\; port = 1080$

$protocol : tcp \;\; udp$

$proxyprotocol : socks\_v4 \;\; socks\_v5$

$method : none \qquad \#username$

}

6.4.3.4 Dante Server

Dante-server is the server part of program Dante. It is a SOCKS server that function as a firewall between networks. It receive connection request coming from SOCKS clients in its internal interface, and forwards the data received from the SOCKS client to the external interface. To start the server, you need to run its daemon /usr/local/sbin/sockd by executing as root the following command:

$$/etc/init.d/danted \;\; start$$

To stop the server, you can execute the following command:

$$/etc/init.d/danted \;\; stop$$

However, some configuration information need to be provided before you fire up the server. These information are provided by editing the configuration file $/etc/danted.conf$ [Kak11].

6.4.3.5 Configuring the Dante-server

Dante-server configuration file contains three main sections: servers settings, rules, and routes [Kak11]. The server settings section provides variables necessary to control things like the location

of the log messages files, the proxy server address and port, the IP address used for all server outgoing connections, and the authentication methods. Authentication methods [Kak11] are a set of rules for authenticating the proxy clients. There are of two types: the clientmethod, and the method. The clientmethod keyword lists the methods needed for specifying the client access rules needed before and during the protocol negotiation phase, while the method keyword lists the authentication methods needed after the completion of the protocol negotiation phase [DA502].

The second section of the configuration file contains the rules. Rules are of two kinds. SOCKS client access rules and SOCKS command rules. SOCKS client access rules is a set of rules that controls the way internal clients connect to the server internal interface. The verification and validation of these rules occur before any data is received from the client application. All such rules are recognized for having the client prefix like in the following example.

$Client\ pass\ \{$

$from : 127.0.0.0/24\ \ port\ \ 1 - 65535\ \ to : 192.128.0.1$

}

The SOCKS command rules is a set of rules that control which external hosts and services can be reached by the internal clients through the server. Unlike the SOCKS clients access rules, these rules don't have the prefix client as shown in the following example.

$pass\ \{$

$from\ : 127.0.0.0/24\ \ to : 0.0.0.0/0$

$protocol\ \ TCP\ \ UDP$

}

The host section of the configuration file contains the routes. The rules in this section govern the way a proxy server chaining can be implemented, if desired [Kak11].

6.4.3.6 An example of Dante-server configuration file [Kak11]

Server Settings

The server will log both via syslog, to stdout and to /var/log/sockd

$logoutput : syslog\ \ stdout\ \ /var/log/lotsoflogs$

$internal : 127.0.0.1\ \ port = 1080$

All outgoing connections from the server will use the IP address

$external :\ \ eth0$

A SOCKS server behind a home router

list over acceptable methods, order of preference. A method not set

here will never be selected. If the method field is not set in a

rule, the global method is filled in for that rule.

Client authentication method:

$method : username\ \ none$

The following is unnecessary if not doing authentication. When doing

something that can require privilege, it will use the userid "sockd".

$\#user.privileged : sockd$

When running as usual, it will use the unprivileged userid of "sockd".

user.notprivileged: sockd

$user.notprivileged : nobody$

Do you want to accept connections from addresses without dns info?

what about addresses having a mismatch in dnsinfo?

$srchost : nounknown\ \ mismatch$

RULES

There are two kinds and they work at different levels.

#================== rules checked first ==================

Allow our clients, also provides an example of the port range command.

$client\ \ pass\ \{$

$from :\ \ 192.168.1.0/24\ \ port\ \ 1 - 65535\ \ to : 0.0.0.0/0$

$\}$

$client\ pass\{$

$from : 127.0.0.0/8\ \ port\ \ 1-65535\ \ to : 0.0.0.0/0$

}

$client\ block\{$

$from : 0.0.0.0/0\ \ to : 0.0.0.0/0$

$log : connect\ \ error$

}

#================== The rules checked next ================

$pass\ \{$

$from : 192.168.1.0/24\ \ to : 0.0.0.0/0$

$protocol : tcp\ \ udp$

}

$pass\{$

$from : 127.0.0.0/8to : 0.0.0.0/0$

$protocol : tcpudp$

}

$pass\ \{$

$from : 0.0.0.0/0\ \ to : 127.0.0.0/8$

$protocol : tcp\ \ udp$

}

$block\ \{$

$from : 0.0.0.0/0\ \ to : 0.0.0.0/0$

$log : connect\ \ error$

}

See /etc/sockd.conf of your installation for additional examples of such rules.

6.4.3.7 Using Dante with Tor

After reviewing the principles governing both Dante-client and Dante-server, and after studying how to configure both of them, it's time to use it in order to connect our client application to Tor. The first idea was to have a schema made of our client application, Dante-client and Tor as shown in the figure 6.3.

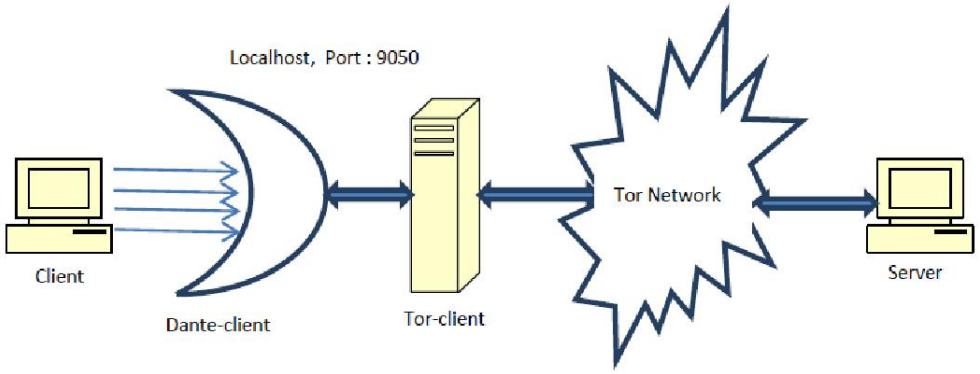

Figure 6.3: Using Dante-client with Tor

We edit Dante-client configuration file to allow it to redirect all connections to Tor. We particularly make sure to tell Dante-client to redirect all connections from the client to the port 9050 of the localhost, which is the address and port where Tor is listening to internal connections. This is done by midifying the variable route in the configuration file as following.

Send client requests to the proxy server at the address shown:

$route\{$

$from : 0.0.0.0/0 \ \ to : 0.0.0.0/0 \ \ via : 127.0.0.1 \ \ port = 9050$

$protocol : tcp \ \ udp \qquad \#server\ supports\ tcp\ and\ udp.$

$proxyprotocol : socks\_v4 \ \ socks\_v5 \qquad \#server\ supports\ sock\ v4\ and\ v5.$

$method : none \qquad \#username\ \#we\ are\ willing\ to\ authenticate\ via$

$\#method\ "none",\ \ not\ "username".$

$\}$

After a giving a try, the result still negative. After several try and several research on the issue, we found out that apparently the Tor client is effectively a client and is not acting like a proxy server. Therefore we need to introduce a SOCKS server in the schema, in this case Dante-server.

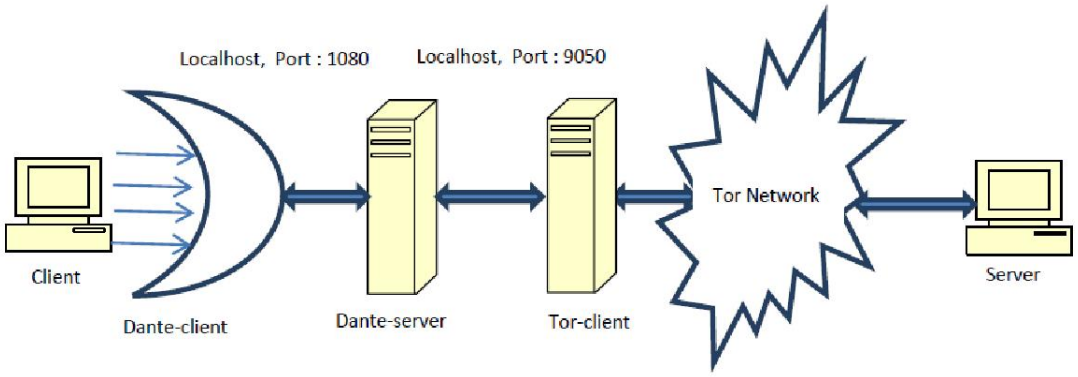

Figure 6.4: Using Dante-server with Tor

One the Dante-server is added into the schema, we needed to change the Dante-client configuration file in order to tell it to redirect all connections to Dante-server instead of Tor as in the previous schema. This is done by modifying the variable route in the configuration file as following.

Send client requests to the proxy server at the address shown:

$route\{$

$from : 0.0.0.0/0 \ to : 0.0.0.0/0 \ via : 127.0.0.1 \ port = 1080$

$protocol : tcp \ udp \qquad \#server \ supports \ tcp \ and \ udp.$

$proxyprotocol : socks\_v4 \ socks\_v5 \qquad \#server \ supports \ sock \ v4 \ and \ v5.$

$method : none \qquad \#username \ \#we \ are \ willing \ to \ authenticate \ via$

$\#method \ "none", \ not \ "username".$

$\}$

Then we will configure Dante-server to redirect the internal connections received from the Dante-client to Tor. The new schema is represented in the figure 6.4. The major modification to the Dante-server configuration file involves the modification of the server settings as following.

Server Settings

The server will log both via syslog, to stdout and to /var/log/sockd

$logoutput : syslog \ stdout \ /var/log/lotsoflogs$

$internal : 127.0.0.1 \ port = 1080$

All outgoing connections from the server will use the IP address

$external : 127.0.0.1 \ port = 9050$

After several try, things still are not working. After investigating on the issue, we found out that in order to redirect the internal connections to Tor, Dante-server needed to be get added a module, called the Redirect module, which is a commercial module sold 200 Euro. Lacking the Redirect module within Dante-server, we were not able to push the experiment with Dante further.

Knowing that we were on the right path with Dante, we start looking for a program similar to Dante but that was completely open source. Our investigation conducted us to discover the program called Socat. After setting Socat, we finally succeeded to connect our client application to Tor by using Socat to redirect the connection through Tor.

Chapter 7

The Implementation

The project program consists of a Linux executable written in C++. It consists of several C++ sources files, together with associated header. The code is mostly original, but we have taken care to keep the acknowledgments to the authors of the original author, together with any copyright notices, inside all code coming from a third party. All cryptographic functions are performed using the Crypto++ library. Crypto++ is a free and open source C++ class library that implements the major cryptographic algorithms. It was written and released in 1995 by Wei Dai. The library us widely used to implement cryptographic functionality into projects in academia, or in both non-commercial and commercial projects. In this program we have used the version 5.6.1.

The program can be divided in two parts: the client and the server. Both the client and the server take advantage of all the other modules implemented in this program. The program is logically divided into five modules, each of which has at least one source code file. We will describe each of these modules in its own section of this chapter. The modules are : the cryptographic modules constituted by the Diffie-Hellman module, the RSA module, and the DES module, the Tools module, and the Tag generation module.

7.1 The client

The client helps the user to perform operations authorized by this protocol. These operations are: to generate, to upload a file to the server, retrieve a file from the server, and delete a file. By now only the tag generation have been implemented. The client generates the tag by collaborating

with another client. Therefore, this protocol necessitate at least two clients on order to work.

7.1.1 The general form of the client

The client functionality is contained in the file MainClient.cpp. It is made of a menu and three functions. The menu offers to the user the possibility of selecting which operation to perform. For this purpose five options are presented to the user. The first option allow the user to generate the tag, the second option allows him to upload a file, the third one allows him to retrieve a file from the server, the fourth option allows him to delete a file, and the last option is responsible to quit the program.

The functions into the client are responsible to perform each of the operations invoked by the options above. These functions are : $CreateTag()$, $Upfile()$, $DownFile()$, and $DelFile()$. The function $CreateTag()$ is responsible for the tag generation, while the responsibility of the function UpFile(0 is to upload a file into the server, and the one for the function $DownFile()$ is to retrieve the file from the server, and the last function, $DelFile()$, is used to delete a file stored on the server.

7.1.2 Client design

The way in which the different components of the client program fit together is as follow. To perform the different operations such as to generate a new tag, to upload a file, to retrieve a file from the server or delete a file, the main program calls a function associated with each of these operations. For instance, to generate a new tag, the client program presents a menu to the user asking him which operation does he want to perform. The user will choose for instance the option 1 which is the one corresponding to the tag generation. This will call the function $CreateTag()$ located in the MainClient.cpp file. This function will call in turn functions that are in the tag generation module. Those functions in the tag generation module are going to call functions that are implemented in the cryptographic modules and the Tools module.

7.2 The cryptographic modules

The cryptographic modules are a set of modules responsible to provide cryptographic tools to our project. Those tools allow us to perform all necessary cryptographic functionality. These modules are: the Diffie-Hellman module, the RSA module, and the DES module. Each module has two files, one for the class definition and another one for it implementation.

7.2.1 The Diffie-Hellman module

The Diffie-Hellman module is used for the implementation of the functionality provided by the Diffie-Hellman algorithm. This module has two files, Dhtool.h as the interface and Dhtool.cpp as the implementation.

7.2.1.1 General form

The two files that constitutes the Diffie-Hellman module are Dhtool.h and Dhtool.cpp. In the Dhtool.h is defined the class $DHGen()$ that implements all the Diffie-Hellman functionality. The members methods are $GenKeys()$, and $genSecret()$. The method $GenKeys()$ is responsible to generate the Diffie-Hellman keys while the method $genSecret()$ is responsible to generate the shared secret that will be hashed to provide the tag.

7.2.1.2 The method GenKeys

The Diffie-Hellman key generation functionality is provided by the $GenKeys()$ method. This method is generating Diffies-Hellman keys and stores them into files for future use. Diffie-Hellman is a key agreement algorithm that allows two clients to establish between them a communication channel that is secure. The function starts with the generation of a safe random prime for use in Diffie-Hellman. Then the function moves to the next step that is, generating Diffie-Hellman parameters. These parameters are p, q, and g, and are such that p = qr + 1. The parameter p is the prime module, while g is considered as the group generator and q is a subgroup order. For the Diffie-Hellman algorithm to work, both clients should use the same parameters. For the simplification of the design, we have hard coded theses parameters.

Any time the parameters are loaded from an outside source, they must be validated in order to avoid to use parameters that have been poorly generated. For this reason we perform a validation on ours hard coded parameters. These three steps, generating a random prime, generating parameters and do their validation are common to both methods $GenKeys()$ and $genSecret()$ implemented in the file DHtool.cpp.

Having all the parameters ready and initialized, we are ready to generate the two keys, the public key, and the private key these two keys are encoded using the HexEncode object provided by Crypto++, and are stored on files for future use. This last step is not necessary if we want these keys to be one time keys. In addition, one may object that storing keys in file may introduce flaw into the algorithm. Although we didn't implement it, the fix could be to use a key sharing algorithm like Shamir key sharing to split the keys into several pieces that will be redistributed and stored to different hosts. When needed, those pieces can be retrieved from their hosts in order to reconstitute the desired key.

7.2.1.3 The method genSecret

The method $genSecret()$ is basically implementing what is know as the Diffie-Hellman key agreement. This method takes two arguments of type string. The first argument is the client private key, and the second argument is the peer public key. This method uses the same parameters generated at the beginning of the $GenKey()$ method, otherwise the key agreement could not be reached.

Since the keys given as arguments to $genSecret()$ are Hex encoded, the first thing we do before to use them is to decode them. The function pursue it execution by initializing a secret key object of type SecByteBlock. We then calculate the secret key based on the client private key and the public key received from the peer client collaborating to the tag generation. After generating the secret key, we can run a hash function on it using md5. The result is the tag. This tag will be encoded using the HexEncode object and stored in the file for future use. This last step is necessary only if we consider the tag as a long term key. Before to use it we have to make sure that both clients agree on the same secret.

7.2.2 The RSA module

This module is used to implement public key cryptography functionality by means of the RSA algorithm. These functionality involve the encryption and decryption process, as well as the digital signature. This module has two files, RSAtool.h for the interface and RSAtool.cpp for the implementation.

7.2.2.1 General form

RSAtool.cpp and RSAtool.h are the two files that constitute the RSA module. In RSAtool.h is defined the class $RSAGen()$ defining all RSA functionality used in this work. The methods that are defined are : the method $RSAES\_genKeys()$, the method $RSASS\_genKeys()$, the method $Encrypt()$, the method $Decrypt()$, the method $Sign()$ and the method $Verify()$. The $RSAES\_genKeys()$ method is responsible for generating both private and public key for encryption scheme, while the method $RSASS\_genKeys()$ will be generating the two keys for digital signature. The method $Encrypt()$ and $Decrypt()$ implements the encryption and the decryption functionality while the methods $Sign()$ and $Verify()$ implements the digital signature and the signature verification.

7.2.2.2 RSA keys generation

In this section we will describe the key generation process using the RSA algorithm. This is done using two different functions : $RSAES\_genKeys()$ and $RSASS\_genKeys()$. The function $RSAES\_genKeys()$ is for generating keys for RSA encryption scheme while $RSASS\_genSecret()$ is generating keys for signature scheme. The reason for two different sets of keys is that the management approaches and the time frames differ for the use of the keys either for signing or for encryption. You may want a signing key to have a long time validity so that peoples that interact with you can check signatures from the past. In opposite, you want to be able to roll an encryption key over as soon as possible. When needed.

Both functions, $RSAES\_genKeys()$ and $RSASS\_genKeys()$ proceed by following the same steps but by using sometimes different objects. They both start by generating a random number rng.

This random number is used to generate RSA parameters needed to initialize both the public and private keys. Next in $RSASS\_genKeys()$ the private key is used to initialize a Signer object while the public key is initializing the Verifier object. The equivalent of this step in $RSAES\_genKeys()$ is the generation if the objects Encryptor and Decryptor. In $RSASS\_genKey()$ the Signer is used to generate the actual private key while the Verifier object is used to generate the actual public key. Both keys are encoded using the HexEncode object and following the Basic Encoding Rules (BER). The situation looks similar in $RSAES\_genKeys()$ where the Encrypt is used to generate the actual public key while the Decrypt object is used to generate the private key. In this case both keys are also encoded using the HexEncode object and following BER. The two sets of keys are then stored on files for future use.

7.2.2.3 RSA encryption

In this section we will describe the encryption process using the RSA algorithm. This function is performed by the method $Encrypt()$ implemented in the file RSAtool.cpp. This method takes two arguments, the first argument being the plaintext to be encrypted, and the second argument is the key used for encryption. We start the process by generating a random number that will be used along with the key size to generate the RSA parameters. The key used for encryption is the public key received from the peer. It was encoded before being transmitted over the network. Therefore we need to decode it before to use it. To do so, we declare a decoder of type HexDecoder, and feed it with the peer public key of type string. The decoded will finally grant us with the actual key of type PublicKey. This is the actual key used for the encryption. The encryption is done using an object $RSAES\_OAEP\_SHA\_Encryption$, initialized with the public key, and a filter. The filter takes the plaintext as source and outputs the ciphertext.

7.2.2.4 RSA decryption

In this section we are going to give a description of the decryption process performed by means of the RSA algorithm. This function is performed by the method $Decrypt()$ implemented in the file RSAtool.cpp along with the encryption method. As with his peer encryption method, this method also takes two arguments. The first argument is ciphertext to be decrypted while the second

argument is the decryption key. The decryption key is the client private key previously generated by mean of the method $RSAES\_genKeys()$.

This algorithm is similar to the encryption method except that it uses a private key instead of a public key used for encryption. The decryption key also need to be decoded because after being generated it was encoded and saved to file. The decryption is done by an object $RSAES\_OAEP\_SHA\_Decryptor$ along with a filter. The object is initialized with the decoded private key while the filter take the ciphertext as input and returns a plaintext as output.

7.2.2.5 RSA digital signing

This section will be dedicated to the description of the digital signature using the RSA algorithm. The RSA digital signing is performed by the method $Sign()$ implemented in the file RSAtool.cpp. This method is given two arguments, with the message to be signed as the first argument and the signing private key as the second argument.

We should not forget that the private key used here for signing is different from the one used for decryption because the first was generated by the method $RSASS\_genKeys()$ while the later was generated by the method $RSAES\_genKeys()$. In this algorithm we generate RSA parameters, and we Hex decode the private key as we have done so in the decryption function. However, we use the private key here to initialize a $RSA :: Signer$ object. The Signer object is used within a SigneFilter filter to generate the Signature. This filter takes the message to be signed as input, and returns its signature as output. The $Sign()$ method returns a value of type Sigmsg, which is a data structure to store a message ans its signature.

7.2.2.6 RSA Signature verification

We describe here the process if verifying a digital signature by means of the RSA algorithm. This is done by the method $Verify()$ implemented in the file RSAtool.cpp. Three arguments are provided to this method. The first argument is the message whose signature need to be verified while the second argument is its signature and the last argument is the public key for signature verification. This key have been generated by the method $RSASS\_genKeys()$.

Like with the method of the previous section, we generate RSA parameters and we Hex decode

the public key provided for signature verification. Unlike the previous method, the public key will be used here to initialize a RSASS::Verifier object. This object will be in turn used along with a filter named SignatureVerificationFilter to verify the signature of the message. The filter takes the message and the signature as input and produces a recovered message as output. If the recovered message from the signature is identical to the original message, then the verification succeeded, otherwise the filter will throw an exception. This method does not return any value.

7.2.3 DES module

This module is used to provide functionality for symmetric key cryptography. These functionality involve the encryption and the decryption processes using DES as well as the hashing process. This module is served by two files, DEStool.h hosting the interface and DEStoo.cpp hosting the implementation.

The Data Encryption Standard (DES) was a previously predominant algorithm for the encryption of electronic data using symmetric cryptography. It is now considered to be insecure for many application because of the size if its key, which is 56-bits long, being considered too small for nowadays computation power. Therefore it has been withdrawn from as a standard for symmetric cryptography. It have been advocated to use either the Advanced Encryption Standard or the Triple DES as its successors. In this work we are using TripleDES in CBC mode.

TripleDES is a short name for the Triple Data Encryption Standard. It is a block cipher algorithm that applies the Data Encryption Standard algorithm three times to each data block. Since the original DES key size of 56 bits became insufficient to protect digital data, TripleDES allows a DES algorithm to increase its key size without the algorithm necessitated to be redesigned completely. It is therefore secure.

Block cipher algorithm can be used in several different modes of operation. In this work we are using TripleDES in CBC mode. A mode of operation is the procedure of enabling the repeated and secure use of a block cipher under a single key [MOV97]. The different modes of operation are ECB, CBC, OFB, CFB, CTR and XTS. An extended explanation about those modes of operation can be found in [MOV97].

In CBC mode, the one used in this work, a XOR operation is applied to each block of plaintext and its predecessor ciphertext. This makes each ciphertext block dependent on all plaintext blocks previously processed. Since the first plaintext block does not have any block ciphertext processed before it, it is given an initialization vector that also helps to get each message unique.

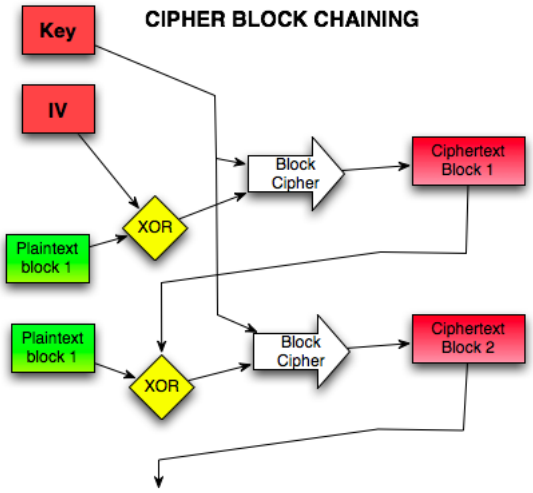

Figure 7.1: CBC mode of operation

The DES module is constituted by the two files DEStool.h and DEStool.cpp. The first file hosts the definition of the class $DESGen()$ while the second hosts its implementation. This class provides three methods, the method $genKeys()$ whose role is to generate the symmetric key that will be used for encryption and decryption, the method $Encrypt()$ responsible of doing the TripleDES encryption, and the method $Decrypt()$ that is performing the TripleDES decryption.

7.2.3.1 TripleDES keys generation

As with the previous keys generation methods, this method starts by generating a random umber prng. This random number is then used to declare and generate the key and the initial vector. Once the key is generated, it is provided to the HexEncoder filter to get encoded. The filter takes the key of SecByteBlock type and returns an encoded key of type string. The initial vector undergoes the same encoding as the key using the same filter, the two values, the key and the initial vector, are then stored on file for future use.

7.2.3.2 TripleDES encryption

In this section we will provide a description of the encryption process using the TripleDES algorithm. This function is enabled by the method $Encrypt()$ implemented in the file DEStool.cpp. The method $Encrypt()$ takes three arguments that are the message to be encrypted as the first argument, the encryption key as second argument, and the initial vector as last argument. The symmetric key and the initial vector have been encoded before being saved on files during the key generation phase, we have to decode them before to use them. These two values are successively provided to a HexDecoder filter to produce their decoded version.

The decoded key and initial vector are then used as parameters to produce an Encryption object of type $CBC\_Mode < DES\_EDE3 >$ that will be used to perform the encryption. This encryption is performed using the Encryption object along with a StreamTransformationFilter filter. This filter takes a string, performs an encryption on it and adds padding if necessary. It produces a string ciphertext as output. The ciphertext is also encoded using an HexEncoder filter, since it will be sent over the network.

7.2.3.3 TripleDES decryption

We are going to describe in this section how the decryption process is done using the TripleDES algorithm. This function is enabled by the method $Decrypt()$ implemented in the file DEStool.cpp. As with the method $Encrypt()$, the method $Decrypt()$ takes three arguments, among which the first is the ciphertex to be decrypted, while the second is the symmetric key, and the last is the initial vector.

All the three arguments of the method $Decrypt()$ underwent an encoding because they had to travel over the network. Therefore we need to decoded them before to use them. We successively pass them to the HexDecoder filer in order to be decoded. The decoded key and initial vector will then serve as parameters to the generation of an object Decryption of type CBC_Mode¡DES_EDE3¿. The object Decryption will be used in the StreamTransformationFilter filter to perform the decryption over a ciphertext given as input, and to produce a plaintext as output.

7.3 The Tools module

There are more functions in our work that cannot be classified into any of the cryptographic module. We have proposed to ourselves to put those functions into a module called Tools module. This set of functions involves function like the ones to save and retrieve keys on file, to send and retrieve either the keys or the half tag to or from the server. In this section we are going to describe these functions.

7.3.1 General form

This module spans over two files, Tools.h that hosts the class $GenTools()$ and the file Tools.cpp that implements that class. The methods that are implemented are the following: $getlocalkey()$, $Send\_pubkey()$, $Send\_keys()$, $Send\_htag()$, $Sendreq()$, $Request\_htag()$.
The method $getlocalkey()$ is used to retrieve a key from a local file while the method $Send\_pubkey()$ is used by the client to send public keys to the server. The methods $Send\_keys()$ is also used to realize the socket connection and to perform the send operation. The method $Send\_htag()$ is used to send the half tag to the server. The method $Sendreq()$ and $Request\_htag()$ are used to request respectively public keys and the half tag from the server.

7.3.2 The method getlocalkey

The method $getlocalkey()$ is used to retrieve a key that is stored in a local file. It takes one arguments, which is the name of the file from which we want to retrieve the key. This method return a key of type string.

7.3.3 The method Send_pubkey

The method $Send\_pubkey()$ is used to send clients public keys to the server. These keys are the ones generated by means of the RSA module, and stored on file. Since the two public keys, one for encryption and another one for digital signature, were stored in two different local files, we need to retrieve them before to send them to the server. Therefore we call twice the method $getlocalkey()$ with the file name as argument indicating every time the name of the file storing each public key.

After retrieving the keys, we provide them as arguments to the function $Send\_keys()$ that takes care of all socket functionality.

7.3.4 The method Send_keys

The method $Send\_keys()$ prepare all the parameters necessary to create a socket connection to the server. The method then prepares two structures, the buffer and the structure Sync. The buffer is going to hold the keys to be sent while the structure Sync will be sent first to the server to notify it about the size and the type of the coming data. By this way the server will reserve enough buffer space and of appropriate type to handle the data coming.

7.3.5 The method Send_htag

The method Send_htag is used by the client to send the half tag to the server. This method takes two arguments, among which the first argument is the data structure to hold the half tag data, and the second is the identification number of the peer client we want to collaborate with in the process of generating a new tag. This method starts by preparing the parameters necessary to establish a socket connection to the server. Once the connection is established, the method will prepare a Sync structure and will send it to the server prior to sending any data. Its purpose is to help the server to be ready to receive the data. If the server fails to receive the data, it will send back an error message to the client.

7.3.6 The method Sendreq

The method $Sendreq()$ is used by the client to request the server to send the peers client to request the server to send the peers public keys, previously uploaded by the peer client. It takes one argument, the identification number of the peer client. The first step of this method is also to prepare the parameters for a socket connection to the server. The next step is to send the Sync structure indicating to the server the nature of the data the client is requesting. The method the get ready to receive the data from the server. The first message coming from the server is a Sync indicating to the client how the buffer space it should allocate for the coming data. After all these

preparations, the method invokes the socket function $receive()$ to receive the data from the server. This function returns a string as output.

7.3.7 The method Request_htag

The client use the method Request_htag to request the peer half tag from the server. It take one argument , which is the identification number of the peer client. This method is similar in its functioning to he method $Sendreq()$ except that in this case, after having prepared the parameters for the socket connection, and after having sent the Sync to the server notifying it about the client desire of receiving the peer half tag, the server will send to the client another Sync to let it know about the size of the data coming. The client will reserve accordingly a buffer of type of the half tag.

7.4 The tag generation module

In this section we are going to describe the different operations performed by the client in order to generate the tag. The tag is the result of a hashing function on a Diffie-Hellman shared secret. Therefore we need the collaboration of the two clients to generate the tag. Since this collaboration of clients takes place offline, we need the support of a server to play the intermediary between the two clients. The functioning id the server will be explained later in this chapter. The tag generation protocol have been described in section 5.2.1 of the chapter 7. This section focuses on showing how we processed and which tools we used in order to materialize each step of the tag generation protocol.

7.4.1 General form

The tag generation module spans over two files that are GenTag.h and GenTag.cpp. The process of tag generation is managed by the class $GenTag()$ defined in the file GenTag.h. This file defines the following methods : $makeHtag()$, $makeTag()$, $UpFile()$, $DownFile()$ and $DeleteFile()$. The method $makeHtag()$ is used to generate the half tag while the method $makeTag()$ is responsible to generate the whole tag. The remaining methods involve respectively the upload of a file

to the server, the retrieval of a file from the server, and the delete of a file stored on the server. Those methods are not yet implemented, therefore we will not discuss them in this work. The implementation of the methods is in the file GenTag.cpp.

Since the generation of the tag will involve a lot of cryptographic operations, we start the process by creating the instances of the classes providing all cryptographic functionality. These classes are $DHGen()$ for all Diffie-Hellman functions, $RSAGen()$ for all RSA functions, $DESGen()$ for all TripleDES functions, and $GenTool()$ for all remaining non cryptographic functions.

7.4.2 The method makeHtag

In this section we describe the encryption phase of the tag generation protocol. This phase end up with the making of a data structure, called here a half tag, containing all the encrypted and signed data. This half tag will be uploaded to the server in order to be sent to the peer client for the decryption phase.

This phase starts with the generation of all keys. These keys include the two keys generated by the Diffie-Hellman algorithm, the two keys generated by the RSA encryption scheme, the two keys generated by the RSA signature scheme, and the symmetric key generated by the TripleDES algorithm. The next step is to send all RSA public keys to the server. To do that, we call the method $Send\_pubkeys()$ provided by the Tools module. After uploading his own RSA public key to the server, it is time for the client to retrieve the peer public key from the server for encryption scheme. To do that, the client will invokes the method $Sendreq()$ provided by the Tools module. After the keys exchange, we can start doing the diverse encryption.

The client will download the DH public key and the RSA private key for signature scheme from locale files using the method $getlocalkey()$ provided by the Tools module. He will then use this RSASS private key to sign the DH public key using the method $Sign()$ provided by the RSA module. The signature of the DH public key is a binary message and will need afterward to be sent over the network. For that reason we encode it using a HexEncode object. The DH public key and its signature need to be encrypted with a symmetric key. For this purpose the client download the TripleDES key and the initial vector from local files, and use them to encrypt the DH public

key along with its signature by mean of the $Encrypt()$ method provided by the DE module. The symmetric key and its initial vector will need to be also sent to the server. Therefore we need to protect them. Using the peer RSAES public key previously downloaded from the server, we encrypt the symmetric key and its initial vector with the method $Encrypt()$ of the DES module. The resulting ciphertext are also encoded using the HexEncode object.

After having all the pieces of data ready, the client will create a data structure called the half tag. This half tag will be filled with the encrypted DH public key, the encrypted signature of the DH public key, the encrypted symmetric key and initial vector, and the identification number of the peer client. All this packet will be sent to the server by mean of the method $Send\_htag()$ provided by the Tool module. The sending operation terminates the encryption phase of the tag generation protocol. At the same time the peer client will go through the same process, and will upload his half tag to the server.

7.4.3 The method makeTag

This section will be focused on the description of the decryption phase of the tag generation protocol. This phase terminates with the generation of the tag. This tag may be saved on file or not depending whether we want to use it as a one time tag or a long term tag.

Before the start of this method, we assume that the method $makeTag()$ have been run and successfully completed. This means that the two clients have successfully uploaded their half tag to the server. This pahse starts with the client retrieving the peer half tag from the server and downloading his RSAES private key from a local file. The peer half tag is retrieved using the method $Request\_htag()$ provided by the Tools module while the RSAES public key is downloaded using the $getlocalkey()$ method provided also by the Tools module.

The decryption process is going in a reverse order of the encryption process performed by the method $makeHtag()$. The last data to be encrypted in the half tag during its making were the symmetric key and its initial vector, and it was encrypted with the peer RSAES public key. Therefore there should be the first to be decrypted. Before do so, we need to decode them using the HexDecode object, since they where encoded after being encrypted. The decryption of the

symmetric key and the initial vector is done with the RSAES private key previously downloaded from a local file, and by using thee method $Decrypt()$ provided by the RSA module. Once the symmetric key and the initial vector are recovered, they can be now used for the decryption of the DH public key and its signature. This decryption is done using the $Decrypt()$ method provided by the DES module. After being decrypted the signature need to be decoded using the HexDecode object, for it was encoded after being encrypted during the execution of the method $makeHtag()$.

The client has now access to the peer DH public key, and he need to verify its integrity before to use it. This is done by the verifying the signature. Since the peer DH public key have been signed with the peer RSASS private key, we need the peer RSASS public key to verify its signature. The peer client had previously uploaded his RSASS public key to the server, therefore we need to retrieve it for the sake of the signature verification. The retrieval is done using the method $Sendreq()$ provided by the Tools module. We then verify the signature using the peer RSASS public key along with the method $Verify()$ provided by the RSA module.

If the peer DH public key pass the test of integrity, it is time to use it. The client will download its DH private key from a local file, and use it along with the peer DH public to generate the DH shared secret. This is done by using the method $genSecret()$ provided by the DH module. The client then apply a md5 hash function on the shared secret to get the tag.

Until now we have described what is happening on the client side. Let see now how the server process its part of the work.

7.5 The server

The architecture we are implementing allows clients to communicate to each other in a offline manner. This is made possible by the use of a server that play the role of intermediary between different clients. The server is designed to fulfill three functions, including storing the RSA public keys, storing the half tag, and finally storing client files in an anonymous manner.

7.5.1 General form of the server

The server is implemented in the file MainServer.cpp. The server may be implemented by using a light database system that will be storing clients data. Light databases like SQLite and Berkeley DB may be good candidate for this function. For the simplification of this design we have implemented the server as a file server. The server stores two different files, including the file Keystore where the clients RSA public keys are stored and the file Htagstore where the clients half tag are stored. Before being stored in the files, clients RSA public keys and clients half tag are first stored respectively in a vector which is in turn stored in the file. Vectors have been preferred to any other structure due its extensible capability. Using vectors, adding new elements in a file can be easily handled.

7.5.2 Server design

Two principal functions are implemented in the file MainServer.cpp, including the function $main()$ and the function $SocketHandler()$. The function $main()$ prepares all the parameters for a socket connection, waiting for any incoming connection from the clients. After initializing a socket connection, this function invokes the function $socketHandler()$, which is an event handler, allowing the server to define what actions to perform when socket events are triggered. This provides threading capability to the server and enables it to handle connections from multiple clients.

The function $socketHandler()$ defines which action to take when the server receive a request from a client. Every request from a client is preceded by the Sync message. Therefore whenever the $socketHander()$ function detects an event, it prepare a buffer space to receive the Sync. After receiving the Sync, it proceeds to retrieve and to interpret the data from the Sync. Those data include the operation, the option, the clientID, the peerID, and the datasize. The clientID and the peerID are the identification numbers of the two clients collaborating in this process. The operation is a one bit value specifying either the server is receiving data (operation = 1) or the server is sending the data (operation = 0). The option is also a one bit value specifying the nature if the data the server is receiving or is sending. The involved data can be the client RSA public keys

(option = 1) or the client half tag (option = 2).

After reading the data from the Sync, the server uses the bits operation and option to decide which action to take. By combining the two operations and the two options, four cases can be envisioned.

7.5.2.1 Case 1: operation = 1, option = 1

In this case, the server is receiving data that it needs to store, and these data are the client RSA public keys. The server starts by preparing a buffer to receive the client RSA public keys, buffer whose size is specified by the field data size from the structure Sync. The server the uses the socket function receive to receive the keys. It then opens the appropriate file in order to store those keys. Before to do so, it needs to check if the given pair (clientID, peerID) has an entry in the file or not. If the pair is already in the file, the server just update the values of the keys, otherwise it will create a new entry in the file and store the keys.

7.5.2.2 Case 2: operation = 1, option = 2

This combination of operation and option means that the server is receiving clients half tag that needs to be stored in file. As in the case 1, the server will prepare a buffer whose size is given by the value of the datasize field from the structure Sync. The server then go ahead and receive the half tag that it will try to store into the appropriate file. Eventually the server will proceed to check the existence of the pair (clientID, peerID) in the file in order to know if it should update an existing entry or it should create a new entry for the half tag.

7.5.2.3 Case 3: operation = 0, option = 1

In this case the server in receiving a request from a client to get the RSA public keys of the peer client whose peerID is given in the structure Sync. The server starts the process by opening the file where keys are stored. It checks for the existing of the pair (clientID, pairID) gotten from the structure Sync. If the pair is not found, the server will send back a error message to the client, otherwise the server prepares a structure Sync that will filled with the size of the retrieved keys. That Sync will be sent to the client in order to notify it about the size of the coming data. Following

the Sync, the keys are put into a buffer and sent back to the requesting client.

7.5.2.4 Case 4 : operation = 0, option = 2

This combination of operation and option values tells the server that the client is requesting a half tag belonging to the peer client whose peerID is in the structure Sync. Using the pair (clientID, peerID) the requesting client identification number and the peer identification number, the server opens the file storing the half tag and checks for the existing of the pair. In case of a negative result the server notify the client, otherwise it retrieves the half tag from the file, and prepares the Sync by filling it with the half tag size. Both the Sync and the half tag are sent following each other, with the Sync announcing to the client the size of the coming half tag.

7.6 The anonymous communication

The client and server implemented in this work are both SOCKS unaware applications. Therefore we need to use a socksifying application if we want to access a SOCKS server. In this section we are going to describe the setting of the anonymous communication using Tor.

Our previous attempts to redirect the communication through the Tor network had shown us that we need to combine two applications to achieve this objective. The first application is a SOCKS client or socksifier, that is making our participating clients to be socks aware. The second application should be a bidirectional application that is able to redirect incoming communication to Tor and forward back Tor response to the client. We have chosen to use Socat for such purpose.

Socat (Socket CAT) [SOC12] is a mculti-purpose network relay that allows a bidirectional data transfer between two independent data channels. Socat accepts different types of data channels including a file, a pipe, a device, a socket, an SSL socket, a proxy CONNECT connection, a file descriptor, and a program. Socat can have various use including as a TCP port forwarder, as a external socksifier, as a tool for attacking weak firewalls, as a shell interface to UNIX sockets, as an IPv6 relay, and as a tool for redirecting TCP oriented programs to a serial line.

Socat has an available rich set of options that can be used to refine its behavior. Those options include a terminal parameters, $open()$ options, file permissions, file and process owners, basic

socket options like bind address, advanced socket options like IP source routing, TTL, and TCP performance tuning. These options, to which it can be added more advanced capabilities including a daemon mode with forking, client address check, some stream data processing, choosing socket, pipes, debug and trace option, precise error messages, make Socat a very versatile network tool.

To obtain the anonymous communication, we have to configure the three components used, Tsocks, Socat and Tor client. We configure Tsocks through tsocks.conf, its configuration file, by editing the following variables.

$$Server = 127.0.0.1 \;\; server\_type = 5 \;\; server\_port = 1080$$

This tells Tsocks to forward all incoming connections to the localhost listening on the port 1080. WE assume that we have Tor already installed and configured to listening to incoming connection on the default $port9050$. If we have the server running on the host kix.netsec.colostate.edu and listening to on port 8000, we can use the following Socat command.

$$SocatTCP4 - LISTEN : 1080, forkSOCKS4A : localhost : kix.netsec.colostate.edu : 8000, socksport = 9050$$

The above command causes Socat to listen to on $port1080$, and tunnel all incoming connections to the host $kix.netsec.colostate.edu(port8000)$ via the the Tor SOCKS server that is listening on $port9050$.

Chapter 8

Conclusion

Over years, we are witnessing a noticeable increase of the amount of personal data collected and stored on servers by simple user or by organizations like hospitals, insurance companies or administrations. This preference for storing data online on dedicated servers is justified by the fact that public servers are reliable and offer a highly availability of data. However, many incidents of security breaches involving an unwilling disclosure of personal information stored on servers have raised concerns among users about this dissemination of personal data. The emergence of new hardware devices that are portable, and have a large enough storage space while they are able to process data securely promises an interesting alternative solution that may allow users to gain control over the management of their personal data. This has lead to the thinking of a Personal Data Server that should allow the user, among others capabilities, a control over the sharing conditions related to her data with a sure way to reinforce theses conditions.

In this work, we propose an implementation of the communication protocol that allows users to anonymously and asynchronously exchange messages and that also supports a secure deletion of messages. We reviewed existing protocols with the networks built around each of them, that are designed to users able to anonymously exchange files. In each of existing solutions that we have investigated, we have identified problems that make them somehow insecure. We have also conducted a detailed analysis of the Tor network, the best anonymizing network among all currently implemented. We then proposed an implementation of the protocols allowing two users willing to share certain data, to generate anonymously an upload-tag. The upload-tag can be used to upload anonymously files that need to be shared, and also to retrieve shared files while also making it

possible to securely delete shared files.

Our implementation allows these protocol to be performed in an anonymous manner. This is made possible by the integration of the Tor network to our design. One of the big challenges was how to redirect all the communications generated by the clients applications, that are non SOCKS enabled, through the Tor client used to access the Tor network? The solution to this problem was to use a socksifier application combined with Socat, a network Swiss knife, to force all communications to go through Tor.

The client and the server have all been developed in C++. The cryptographic library used for this implementation is Crypto++. The server is implemented as a multi-threaded file server application that receives client RSA public keys, client half tag, and put them in a vector that is stored after in a file. The server is also responsible for providing any asynchronous communication needed by any participating client.

An evaluation of the impact of the anonymous communication on the overall performance have been conducted in [RBS+12]. The objective of the measurement was to study the scalability of the anonymous communication implementation using Tor. It was found during this study that the Tor network had a fairly constant overhead except at the first instant of the communication where the overhead was really increased. This is due to Tor performing a circuit connection setup.

The current implementation of the anonymous communication uses a socksifier application and Socat to gain access to the Tor network. This makes our implementation strongly dependent of external applications. We plan to refines this anonymous communication by fully integrating a module allowing it to directly access the Tor network without any help from external an application. This involves the design of a library allowing to manipulate the Tor control port of the Tor client.

Besides providing a non-linkability between a client and its stored files on the server, this implementation allows the server to store these files after being encrypted. Each query of the data involves its decryption. This adds an extra overhead to the total overhead induced by the use of all these cryptographic operations. We want to investigate eventual solutions, like homomorphic encryption schemes, that may allow the query of encrypted data without necessitating its prior decryption.

Although the Tor network is a latency network, it still adds some significant delay to the normal delay of the communication performed by all protocols. It would be interesting to investigate an alternative to Tor. Candidates that need to be studied closely include network communication like JAP [Pro], UltraSurf [Ult], and Freegate [Glo]. A comparative study about the delay induced by each network above is also needed.

REFERENCES

[AJP95] Marshall D. Abrams, Sushil G. Jajodia, and H. J. Podell. *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1995.

[ALLP07] Tim Abbott, Katherine Lai, Michael Lieberman, and Eric Price. Browser-based attacks on tor, 2007.

[BMG+07] Kevin Bauer, Damon McCoy, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Low-resource routing attacks against tor. In *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, WPES '07, pages 11–20, New York, NY, USA, 2007. ACM.

[CC05] Tom Chothia and Konstantinos Chatzikokolakis. A survey of anonymous peer-to-peer file-sharing. In *Proceedings of the 2005 international conference on Embedded and Ubiquitous Computing*, EUC'05, pages 744–755, Berlin, Heidelberg, 2005. Springer-Verlag.

[Cha81] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, February 1981.

[Chr08] Juliusz Chroboczek. The polipo manual, v. 1.0.4., 2008. http://www.pps.jussieu.fr/ jch/software/cpc/cpcmanual.pdf.

[Clo08] Shaun Clowes. Tsocks, 2008. http://tsocks.sourceforge.net/index.php.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *INTERNATIONAL WORKSHOP ON DESIGNING PRIVACY ENHANCING TECHNOLOGIES: DESIGN ISSUES IN ANONYMITY AND UNOBSERVABILITY*, pages 46–66. Springer-Verlag New York, Inc., 2001.

[DA99] Tim Dierks and Christopher Allen. The tls protocol. Technical report, 1999.

[DA202] Dante a free socks implementation., 2002. http://www.inet.no/dante.

[DA302] Commercial dante modules, 2002. http://www.inet.no/dante/module.html.

[DA402] Minimal client configuration, 2002. http://www.inet.no/dante/doc/latest/config/client.html.

[DA502] Minimal server configuration, 2002. http://www.inet.no/dante/doc/latest/config/server.html.

[DD02] John Douceur and Judith S. Donath. The sybil attack. page 251260, 2002.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644 – 654, nov 1976.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *In Proceedings of the 13 th Usenix Security Symposium*, 2004.

[EDG09] Nathan S. Evans, Roger Dingledine, and Christian Grothoff. A practical congestion attack on tor using long paths. In *Proceedings of the 18th conference on USENIX security symposium*, page 3350, 2009.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[Glo] Global Internet Freedom Consortium. Freegate. Available at http://www.internetfreedom.org/FreeGate (Last accessed March 20, 2012).

[Gol01] Oded Goldreich. *Foundations of Cryptography, Volume 1: Basic Tools*. Cambridge University Press, New York, NY, USA, 1st edition, 2001.

[GSB02] Mesut Gnes, Udo Sorges, and Imed Bouazizi. Ara - the ant-colony based routing algorithm for manets, 2002.

[Kah96] David Kahn. *The Codebreakers : The Story of Secret Writing*. New York: Scribner, New York, NY, USA, 1st edition, 1996.

[Kak11] Avinash Kak. Lecture notes on computer and network security, 2011. https://engineering.purdue.edu/kak/compsec/Lectures.html.

[Lee] Ying-Da Lee. Socks : A protocol for tcp proxy across firewalls. http://ftp.icm.edu.pl/packages/socks/socks4/SOCKS4.protocol.

[LGL+96] M Leech, M Ganis, Y Lee, R Kuris, and D Koblas. Rfc 1928: Socks protocol version 5, 1996.

[LLY+09] Zhen Ling, Junzhou Luo, Wei Yu, Xinwen Fu, Dong Xuan, and Weijia Jia. A new cell counter based attack against tor. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 578–589, New York, NY, USA, 2009. ACM.

[LP03] Y. Liu and B. Plale. Survey of publish subscribe event systems, 2003.

[MD05] S.J. Murdoch and G. Danezis. Low-cost traffic analysis of tor. In *Security and Privacy, 2005 IEEE Symposium on*, pages 183 – 195, may 2005.

[MN09] Elisa Bertino Mohamed Nabeel, Ning Shang. Privacy-preserving filtering and covering in content-based publish subscribe systems. Technical report, Purdue University, 2009.

[MOV97] A. J Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FA, USA, 1st edition, 1997.

[PASM08] Vasilis Pappas, Elias Athanasopoulos, Ioannidis Sotiris, and Evangelos P. Markatos. Compromising anonymity using packet spinning. In *In Proceedings of the 11th Information Security Conference*, 2008.

[Pie00] Henna Pietilinen. Elliptic curve cryptography on smart cards, 2000.

[Pro] Project : AN.ON - Anonymity Online. JAP Anonymity and Privacy. Available at http://anon.inf.tu-dresden.de/index_en.html (Last accessed March 20, 2012).

[Pub10] Publishsubscribe pattern, 2010. http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern.

[Ray10] Indrajit Ray. Cs556-computer security, Fall 2010. Computer Science Department, Colorado State University.

[RBS+12] Indrajit Ray, Kirill Belayev, Mikhail Strizhov, Dieudonne Mulamba, and Rajaram Mariappan. Secure logging as a service. delegating log management to the cloud. In *IEEE Security and Privacy*, page (under review), 2012.

[RR98] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for web transactions. *ACM Trans. Inf. Syst. Secur.*, 1(1):66–92, November 1998.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[RSG98] M.G. Reed, P.F. Syverson, and D.M. Goldschlag. Anonymous connections and onion routing. *Selected Areas in Communications, IEEE Journal on*, 16(4):482 –494, may 1998.

[Sal10] Juha Salo. *Recent Attacks On Tor*. PhD thesis, 2010. http://www.cse.hut.fi/en/publications/B/11/papers/salo.pdf.

[Sch95] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1995.

[SER12] Socks servers, 2012. http://en.wikipedia.org/wiki/SOCKS#SOCKS_Servers.

[Sin00] Simon Singh. *The Code Book : The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. New York : Anchor Books, New York, NY, USA, 1st edition, 2000.

[SL05] Mudhakar Srivatsa and Ling Liu. Securing publish-subscribe overlay services with eventguard. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 289–298, New York, NY, USA, 2005. ACM.

[SL07] Mudhakar Srivatsa and Ling Liu. Secure event dissemination in publish-subscribe networks. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, pages 22–, Washington, DC, USA, 2007. IEEE Computer Society.

[SM04] Steve Bono. Christopher A Soghoian. and Fabian Monrose. Mantis: A high performance, anonymity preserving, p2p network. Technical report, Johns Hopkins University Information Security Institute, 2004.

[SOC12] Socat, 2012. http://www.destunreach.org/socat/.

[Sta11] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, Boston, MA, USA, 1st edition, 2011.

[STRL01] Paul Syverson, Gene Tsudik, Michael Reed, and Carl Landwehr. Towards an analysis of onion routing security. In *INTERNATIONAL WORKSHOP ON DESIGNING PRIVACY ENHANCING TECHNOLOGIES: DESIGN ISSUES IN ANONYMITY AND UNOBSERVABILITY*, pages 96–114. Springer-Verlag New York, Inc., 2001.

[Til02] Henk C. A. van Tilborg. *Fundamentals of Cryptology: A Professional Reference and Interactive Tutorial*. Kluwer Academic Publishers, New York, NY, USA, 1st edition, 2002.

[Ult] Ultrareach Internet Corporation. UltrasurfPrivacy. Security. Freedom. Available at http://ultrasurf.us/index.html (Last accessed March 20, 2012).

[WALS08] Matthew K. Wright, Micah Adler, Brian Neil Levine, and Clay Shields. Passive-logging attacks against anonymous communications systems. *ACM Trans. Inf. Syst. Secur.*, 11(2):3:1–3:34, May 2008.

[WCEW02] C. Wang, A. Carzaniga, D. Evans, and A. Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9 - Volume 9*, HICSS '02, pages 303–, Washington, DC, USA, 2002. IEEE Computer Society.

[WIK12] Socks, 2012. http://ftp.icm.edu.pl/packages/socks/socks4/SOCKS4.protocol.

[Win05] Brian J. Winkel. *The German Enigma Cipher Machine: Beginnings, Success, and Ultimate Failure*. Boston: Artech House, Boston, MA, USA, 1st edition, 2005.