2 | Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Privacy-enhancing Technologies for Private Services

von Karsten Loesing

UNIVERSITY OF
BAMBERG
PRESS

Schriften aus der Fakultät
Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

**Schriften aus der Fakultät
Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg**

Band 2

University of Bamberg Press 2009

Privacy-enhancing Technologies for Private Services

von Karsten Loesing

University of Bamberg Press 2009

Bibliographische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Informationen sind im Internet über http://dnb.ddb.de/ abrufbar.

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität als Dissertation vorgelegen.
1. Gutachter: Prof. Dr. Guido Wirtz
2. Gutachter: Prof. Dr. Udo Krieger
Tag der mündlichen Prüfung: 24. April 2009

Herstellung und Druck: Digital Print Group, Erlangen
Umschlaggestaltung: Dezernat Kommunikation und Alumni, Teresa Zak

http://www.uni-bamberg.de/ubp/

ISSN: 1867-7401
ISBN: 978-3-923507-45-0 (Druckausgabe)
URN: urn:nbn:de:bvb:473-opus-1832

To my parents Karin and Wiard

who always give me the safety
that they would catch me if I fell.

Acknowledgments

I want to thank my supervisor, Prof. Dr. Guido Wirtz, for supporting my dissertation even at a time when privacy on the Internet is highly controversial in the German public opinion—which is sad in its own way. Thanks to Prof. Dr. Udo Krieger and Prof. Dr. Andreas Henrich for their support as members of my dissertation committee. I am also indebted to my colleagues Jens Bruhn and Sven Kaffille for uncountable discussions and (hopefully) mutual motivation. Thanks to all students at the University of Bamberg who have contributed to my dissertation project by working on software projects or writing their theses on related subjects, including Knut Hildebrandt, Christian Wilms, Maximilian Röglinger, Domenik Bork, and Jörg Lenhard. Special thanks to my parents for continuous motivation and for proofreading the final manuscript. Also, I thank Annika Putz for her useful comments on the draft of my thesis.

I further thank the people of the Tor project: Roger Dingledine for continuously discussing my ideas to change/improve Tor hidden services, for keeping me motivated to continue my work, and for giving me very helpful comments on the final draft of my thesis. Nick Mathewson for reviewing my patches and giving me useful feedback to improve them; Paul Syverson for giving me invaluable advice on a draft of this thesis; Lasse Øverlier and Steven J. Murdoch for various discussions on Tor hidden services; Peter Palfrader, Matt Edman, Andrew Lewman, and Jacob Appelbaum for their support during Tor development and helping me secure my Tor directory authority; Sebastian Hahn and Jens Kubieziel for their feedback and corrections on one of the papers that is part of the contribution of this thesis.

Zusammenfassung

Privatsphäre im Internet wird immer wichtiger, da ein zunehmender Teil des alltäglichen Lebens über das Internet stattfindet. Internet-Benutzer verlieren die Fähigkeit zu steuern, welche Informationen sie über sich weitergeben oder wissen nicht einmal, dass sie dieses tun. Datenschutzfördernde Techniken helfen dabei, private Informationen im Internet zu kontrollieren, zum Beispiel durch die Anonymisierung von Internetkommunikation. Bis heute liegt der Fokus bei datenschutzfördernden Techniken hauptsächlich auf dem Schutz von Anfragen an öffentliche Dienste. Diese Arbeit wirft die Frage nach den Risiken beim Betrieb von Internetdiensten durch Privatpersonen auf. Ein Beispiel hierfür sind Instant-Messaging-Systeme, die es ermöglichen, Anwesenheitsinformationen und Textnachrichten in Echtzeit auszutauschen. Üblicherweise schützen diese Systeme die Anwesenheitsinformationen, die auf zentralen Servern gespeichert werden, nicht besonders. Als Alternative verringern dezentrale Instant-Messaging-Systeme dieses Problem, indem Privatpersonen sich gegenseitig Dienste anbieten. Allerdings bringt das Anbieten eines Dienstes als Privatperson im Vergleich zu Organisationen oder Unternehmen neue Sicherheitsprobleme mit sich: Erstens werden durch die Verfügbarkeit eines solchen Dienstes Informationen über die Präsenz des Dienstanbieters preisgegeben. Zweitens soll der Standort des Servers unerkannt bleiben, um nicht den Aufenthaltsort des Dienstanbieters zu offenbaren. Drittens muss der Server besonders vor unautorisierten Zugriffsversuchen geschützt werden.

Diese Arbeit schlägt die Nutzung von pseudonymen Diensten als Baustein von privaten Diensten vor. Pseudonyme Dienste verbergen den

Standort eines Servers, der einen bestimmten Dienst anbietet. Der hier geleistete Beitrag soll herausfinden, welche Teile von pseudonymen Diensten, besonders von Tor Hidden Services, fehlen, um sie für private Dienste einzusetzen. Dies führt zu drei Hauptproblemen, zu denen Lösungen vorgeschlagen werden: Erstens skalieren bisherige Ansätze für pseudonyme Dienste nicht für die in Zukunft zu erwartende Anzahl von privaten Diensten. Diese Arbeit schlägt einen neuen Ansatz vor, der Hidden-Service-Beschreibungen in einer verteilten Datenstruktur ablegt, anstatt sie auf zentralen Servern zu speichern. Ein besonderer Fokus liegt auf der Unterstützung von privaten Einträgen, die für private Dienste benötigt werden. Zweitens geben pseudonyme Dienste während des Anbietens im Netzwerk und der Verbindungsherstellung durch Clients zu viele Informationen über die Identität des Dienstes preis. Der in dieser Arbeit verfolgte Ansatz ist, die Informationen, die ein Dienst im Netzwerk bekanntgibt, auf ein Minimum zu reduzieren und nicht-autorisierte Clients am Zugriff auf den Dienst schon während der Verbindungsherstellung zu hindern. Diese Änderungen schützen die Aktivität und das Nutzungsmuster des Dienstes vor nicht-autorisierten Personen. Drittens weisen pseudonyme Dienste eine schlechtere Effizienz auf als Dienste, auf die direkt zugegriffen wird. Der Beitrag dieser Arbeit ist, die Effizienz zu messen, mögliche Probleme zu identifizieren und Verbesserungen vorzuschlagen.

Summary

Privacy on the Internet is becoming more and more important, as an increasing part of everyday life takes place over the Internet. Internet users lose the ability to control which information they give away about themselves or are even not aware that they do so. Privacy-enhancing technologies help control private information on the Internet, for example, by anonymizing Internet communication. Up to now, work on privacy-enhancing technologies has mainly focused on privacy of users requesting public services. This thesis introduces a new privacy risk that occurs when private persons run their own services. One example is instant messaging systems which allow users to exchange presence information and text messages in real time. These systems usually do not provide protection of presence information which is stored on central servers. As an alternative, decentralized instant messaging system designs mitigate this problem by having private persons provide instant messaging services to each other. However, providing a service as a private person causes new security problems as compared to providing a service as an organization or enterprise: First, the presence of such a service reveals information about the availability of the service provider. Second, the server location needs to be concealed in order to hide the whereabouts of a person. Third, the server needs to be specifically protected from unauthorized access attempts.

This thesis proposes to use pseudonymous services as a building block for private services. Pseudonymous services conceal the location of a server that provides a specific service. The contribution made here is to analyze what parts of pseudonymous services, in particular Tor hidden services, are missing in order to apply them for private services. This analysis leads to three main problems for which solutions are proposed: First, known pseudonymous service designs do not scale to the expected number of private services which might be provided in the future. This thesis proposes a new approach to store hidden service descriptors in a distributed data structure rather than on central servers. A particular focus lies on the support of private entries which are required for private services. Second, pseudonymous services leak too much information about service identity during advertisement in the network and connection establishment by clients. The approach taken in this thesis is to reduce the information that a service publishes in the network to a minimum and prevent unauthorized clients from accessing a service already during connection establishment. These changes protect service activity and usage patterns from non-authorized entities. Third, pseudonymous services exhibit worse performance than direct service access. The contribution of this thesis is to measure performance, identify possible problems, and propose improvements.

Contents

List of Tables

List of Figures

1 Introduction

Privacy on the Internet is increasingly becoming a problem. People are using the Internet for their everyday activities, but many do not realize how much information they give away about themselves. Internet users leave traces of every search for information, every email they send, every forum entry they write, and so forth. Companies have long discovered the value of private information for marketing purposes or to classify their customers. Governments are just about to deploy systems to collect private information to better control what their citizens are doing on the Net. Privacy means that people are able to control which personal information they give away and which not. Even though this property is often taken for granted in everyday life, the Internet and its applications make it increasingly harder to protect individual privacy.

The initial motivation for this thesis originates from privacy issues with *instant messaging* systems. The basic operation of an instant messaging system is to disseminate information about its users' presence statuses and permit exchange of text messages in real time. Both presence status and text messages are privacy-relevant, but so far all approaches on protecting instant messaging focus on privacy of text messages. As an example, Off-The-Record messaging [5] extends existing instant messaging systems by encrypting text messages from end to end. The approach does not only conceal message contents, but ensures that contents of a communication session are "off the record" in the sense that neither of the participants can prove that any statement has been made by their communication partner and not by themselves.

However, presence status information is certainly privacy-relevant, too.

One can learn a lot about a person's behavior from keeping track of her presence status. On the one hand, this information needs to be given to intended recipients. On the other hand, nobody else should learn about a person's presence status. What if an instant messaging protocol leaks presence status to unauthorized entities? Does the system ensure that users which are removed from the list of communication partners do not learn about presence status anymore? Could the instant messaging service providers record their users' presence status and pass presence profiles on to third parties?

When looking at instant messaging systems, the main problem of protecting presence information results from their system architectures. Typically, users log in on central servers which provide them with presence statuses of communication partners and announce to them when a user has entered the system. With such a central component it is impossible to guarantee that presence information is only given to intended recipients. In contrast to this, decentralized approaches based on peer-to-peer designs are more likely to solve the given problem. In such a design, there is little or no infrastructure, but users provide the instant messaging service on their own. Unfortunately, a decentralized approach generates a new problem: Users need to establish connections to each other in order to exchange presence information or text messages. But as soon as users reveal their IP addresses, their communication partners could exploit this knowledge to derive private information about the user later on. Any future communication that is directed to or originates from that IP address leaks presence of the user.

The specific problem of privacy-aware instant messaging can be generalized to all kinds of Internet services. In fact, any service which is provided by a private person has specific security requirements that exceed those of usual Internet services. The problem is that the service might reveal information about its provider which are private and shall therefore be protected. This information includes the IP address of the computer on which the service is provided: Anyone can look up in public databases to which network an IP address belongs and where the user's computer is located. Guha and Francis [26] have successfully tracked locations of people by observing changing IP addresses of services provided on laptops. Further, anyone who knows the IP address of a service could try to mount an attack on the service which the user's computer is unlikely to withstand. And finally, service activity might give hints on the presence of the user. These hints can reveal personal behavior or the timezone in which a person resides. Usually, private persons who provide an Internet service on their computer do not wish to make it available to the public, but only to a limited set of users. These requirements make it necessary to protect the service from unauthorized access attempts. In the following, a service that is provided by a private person rather than an organization is referred to as a *private service*. The requirements to private services are:

- The *location* of the server providing a private service is not revealed.
- *Service activity* of a private service is only known to authorized clients.
- Unauthorized clients cannot make any *access attempts* to a private service.

An important part of the solution for the given problems are *privacy-enhancing technologies* [20, 22, 24]. These technologies attempt to give back control over private information to the private persons. An important part of these technologies are *anonymous communication networks* which permit users to communicate without revealing their IP address. In particular, anonymous communication networks enable their users to hide from others to whom they send messages or from whom they receive replies. These technologies are useful to request a service with potentially controversial contents without anyone, including the service provider, knowing who sent the request.

A subset of anonymous communication systems also supports providing a service without anyone learning about the provider's identity. These services are accessed by a pseudonymous address which cannot be linked to the IP address of the server providing the service. This feature is referred to as *pseudonymous services*. Hiding the location of a service is a necessary prerequisite in the attempt to hide service activity and protect a service from attacks. However, the current designs of pseudonymous services are not sufficient to meet all security requirements of private services. Even though the IP address of a pseudonymous service is hidden, service activity is still leaked and services are still vulnerable to attacks. Pseudonymous services have not been designed with the scenario of private services in mind.

Contribution. The contribution of this thesis is to identify what parts of pseudonymous services need to be extended in order to support private services and to propose a working design for the necessary changes. A comparison of pseudonymous service designs will reveal that Tor hidden services [11] are a useful basis for these extensions. Tor is actively used by hundreds of thousands users, has an active community, and the Tor developers are open to discuss changes and accept patches if proven to be useful.

Tor provides anonymity by relaying traffic over a series of nodes to hide the relation between initiator and responder. The initiator therefore builds a *circuit* between her own computer and a series of usually three Tor relays. All messages are encrypted in layers, so that none of the relays can link message content to the initiator. Tor hidden services make use of circuits to provide pseudonymous services. Besides, hidden services promise to resist censorship and distributed denial-of-service attacks. Hidden services are implemented by using a random Tor relay as *rendezvous point*. Both initiator and responder build circuits to this rendezvous point in order to hide their own identity. Rendezvous points are only used for a single connection between a client and a hidden server. In order to accept client requests containing the address of a rendezvous point, a hidden service picks a set of Tor relays as *introduction points*. These work similarly to rendezvous points, but only transfer a single message containing the connection request from client to hidden server. The hidden server makes a hidden service available by publishing a *hidden service descriptor* containing a signed list of introduction points. These descriptors are stored on a set of directory servers from where they can be downloaded by clients. Clients establish a connection by setting up a rendezvous point and sending an introduction request to one of the service's introduction points. Upon receipt, the hidden server establishes the connection using the specified rendezvous point.

Hidden services have not been designed for private services. There are at least three problems of Tor hidden services which need to be improved: First, the hidden service design does not scale to the expected number of private services which might be provided in the future. Second, the hidden service protocol does not hide service availability or prevent unauthorized access attempts which are required for private services. And third, performance of hidden services needs to be improved in order to become more attractive for users, including applications that are based on private services.

The first contribution of this thesis will be to make hidden services more scalable. The current hidden service design is sufficient for a limited number of public services which are available most of the time. But in contrast to public services, availability of private services is likely to change, so that the services need to be made available quite often. These new usage characteristics put significant load on the directory system. The contribution of this thesis is a new approach to store hidden service descriptors in a distributed data structure rather than on central servers. A particular focus lies on the security properties of such a distributed approach and on the support of private entries which are required for private services.

The second contribution is to support client authorization as part of the hidden service protocol. Tor hidden services leak information about the pseudonymous identity of a service and propagate service activity to multiple places in the Tor network. While this is acceptable for public services, private services require activity to be hidden from anyone but authorized clients. The approach taken in this thesis is to stop unauthorized connection requests as an integrated component of the hidden service protocol. Unauthorized clients are not allowed to download a hidden service descriptor or even learn about its existence.

The third contribution is to measure and improve performance of Tor hidden services. Double indirection of requests by means of rendezvous and introduction points results in significant delay during connection establishment. In addition to that, relaying messages over a series of relays with possibly very different performance properties further increases the delay. The contribution is to investigate what parts of the hidden service protocol take most of the time and to propose improvements.

Dissertation Project. Special focus of this thesis is to present a practical design and evaluate it in a realistic environment. In addition to analyzing problems on a conceptual level and proposing a novel design that overcomes these problems, certain efforts have been made to specify and implement the necessary changes in the Tor software. Six proposals containing Tor design changes [31, 36–38, 42, 43] have been submitted and accepted by the Tor project. At the time of writing this thesis most of these changes have been deployed in either stable or development versions of the Tor software. This approach allows to integrate community feedback as well as to evaluate the new designs on a wide scale.

During this dissertation project, a number of peer-reviewed conference papers [39–41] and technical reports [12] have been published by the author in the immediate context of this thesis. The work has also been discussed with the research community during short talks on PET 2007, PET-CON 2007, 2008.1, and 2008.2, and a peer-reviewed talk on HOT-PETs 2008 [45]. Two practicals with a total number of 16 students have been held at the University of Bamberg in 2005 and 2007. Two diploma theses [28, 81] and one bachelor thesis [35] were written in the context of this dissertation project between 2006 and 2008. Some part of the implementation is based on work created during the Google Summer of Code 2007 program[1] and during a project funded by the NLnet foundation[2].

Outline. The next two chapters cover the necessary background that is required to understand the contribution of this thesis. Chapter 2 gives an overview of pseudonymous services. First, some definitions are necessary to obtain a common understanding what pseudonymous services are. These definitions include terms from the areas of distributed systems, computer networks, cryptography, and privacy-enhancing technologies. After that, existing technologies are discussed which either support pseudonymous services or which have contributed to the development of later pseudonymous service designs. Chapter 3 gives more detailed background on the design of Tor hidden services [11]. The chapter describes circuit creation, the directory system, the hidden service protocol, and Tor's threat model.

The following three chapters contain the contribution of this thesis as described above. Chapter 4 describes the distributed descriptor storage, Chapter 5 presents the extension of Tor hidden services towards client authorization, and Chapter 6 covers performance measurements and improvements. It was attempted to write these three chapters so that they are

[1] See the accepted project application: `http://code.google.com/soc/2007/eff/appinfo.html?csaid=33D2740B403CC323` (last checked: Dec 17, 2008)

[2] See the project homepage: `https://www.torproject.org/projects/hidserv.html` (last checked: Dec 17, 2008)

self-contained and can be read independently. They only presume knowledge of the background chapters. All three chapters start with a short problem statement, discuss previous work on the topic, and present the contribution including possible evaluations.

The last two chapters conclude the thesis. Chapter 7 describes work that is related to the contribution. Related work includes approaches to make Tor hidden services more private, various attacks on either revealing the location of hidden services or making them unavailable, and proposed applications based on hidden services. Finally, Chapter 8 concludes the thesis and gives an outlook on future work.

2 Background on Pseudonymous Services

Pseudonymous services are, roughly speaking, services that are accessed via an anonymous communication system using pseudonymous identifiers rather than addresses that can be linked to the service provider. The main intention of setting up a pseudonymous service is to protect the service provider from being identified and made responsible for the provided service. Pseudonymous services are an important building block for private services as motivated in the last chapter. Hiding the location of a private person's computer that provides a service is the first step in hiding the person's activity and protecting the computer from attacks. This chapter gives the necessary background on pseudonymous services.

The next section gives definitions of the properties of pseudonymous services and related concepts. These definitions include terms from the areas of distributed systems, computer networks, security and cryptography, and privacy-enhancing technologies. These definitions help derive a working definition for pseudonymous services that is used throughout this thesis. In the subsequent two sections, existing technologies are discussed that either provide pseudonymous services or that made important contributions to later designs which do provide this feature. In Section 2.2, technologies are presented that permit high-latency recipient pseudonymity, that is, receiving messages using a pseudonym. These could be considered to be the precursors of technologies providing pseudonymous services. The major drawback for using these technologies for pseudonymous services is their intended design to delay messages for hours to prevent traffic analysis attacks. Section 2.3 contains low-latency designs for servers making use of responder pseudonymity, that is, designs for pseudonymous services. Their low-latency properties allow the execution of interactive services as they are required for private services. The discussion of technologies explicitly excludes designs to achieve anonymous storage. These systems provide protection for users storing and retrieving files, but do not support users in running interactive services. Goldberg periodically publishes the state of the art of privacy-enhancing technologies for the Internet [20, 22, 24], including some of those designs that had to be excluded here.

2.1 Definition of Pseudonymous Services

The discussion on background of pseudonymous services requires a few definitions of necessary terms. First, a definition is given for *services* coming from the areas of distributed systems and computer networks. Next, some important security properties are defined that are common for services being distributed over insecure networks in general. Finally, anonymity and *pseudonymity* are defined as properties of privacy-enhancing technologies.

2.1.1 Distributed Systems

The foundation for talking about pseudonymous services is the notion of a service in the context of distributed systems. Coulouris and others [7] define a *distributed system* "as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages." The central point in their definition is the necessity to pass messages over a computer network. As a result, all systems that require only a single computer to execute are excluded from the definition. The authors mention a number of consequences that arise from their definition: First, a distributed system needs to cope with concurrent program execution which is not necessarily the case in non-distributed systems. Second, a distributed system does not have a global clock that could be used to synchronize programs which are executed on distinct networked computers. And third, each component of a distributed system can fail independently, while other components keep running. These limitations need to be taken into account when designing applications for distributed systems.

The above definition does not motivate the reasons for building a distributed system in the first place. The authors give the main motivation for building distributed systems and running distributed applications on them separately: “The motivation for constructing and using distributed systems stems from a desire to share resources.” These resources include both hardware resources, such as disks and printers, and software resources, like files, databases and data of all kinds. In these cases, the main focus of distribution is to access a remote resource to request information from it or change its state. When considering communication applications, the notion of a resource could also be extended to retrieve information about the presence state of a communication partner and the ability to deliver messages.

Passing messages between networked computers may be the defining element for distributed systems. But this definition is not sufficient to describe how a distributed system works. The authors therefore use the notion of a *service* as the means of sharing resources between networked computers. The authors define the term service as “a distinct part of a computer system that manages a collection of related resources and presents their functionality to users and applications.” Presenting the functionality of a service requires a defined interface containing a set of permitted operations. In case of a networked service this interface is provided to other computers by means of exchanging messages. As a result, the operations defined in a service interface constitute the only way of accessing a resource and changing its state.

In the context of providing and accessing a service, there are usually two roles involved: client and server. Tanenbaum and van Steen [78] define these two terms as follows: "A *server* is a process implementing a specific service, for example, a file system service or a database service. A *client* is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply." This definition implies a strict separation of the two roles of server and client with the server being available all of the time whereas clients only need to be available while requesting the service.

In contrast to this, Kurose and Ross [34] give a broader definition of server and client that includes the development of peer-to-peer (P2P) architectures: "In a client-server architecture, there is an always-on host, called the *server*, which services requests from many other hosts, called *clients*. [...] In a P2P architecture, there is minimal (or no) reliance on always-on infrastructure servers. Instead the application exploits direct communication between pairs of intermittently connected hosts, called *peers*." Peer-to-peer architectures do not distinguish as strictly between the two roles of client and server. A peer can act as either client or server depending on the communication context: "In the context of a communication session between a pair of processes, the process that initiates the communication [...] is labeled as the *client*. The process that waits to be contacted to begin the session is the *server*."

In the context of this thesis, a *private service* is one that is provided by a server which is owned by a private person. Usually, the computer that runs the server is not dedicated to perform only this task, and therefore the service is not necessarily available all the time. As a result, activity of the service might correlate with the personal behavior of the service provider.

The way how client and server exchange messages in order to implement and use a service is defined in a *protocol*. According to Kurose and Ross [34], "a protocol defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event.” Schneier [71] further adds the requirement that a protocol must be “designed to accomplish a task.” Protocols constitute a formalization of the behavior of communicating entities to a level that allows evaluation of the non-functional properties of a service including security properties.

2.1.2 Security Properties

The fact that clients and servers need to exchange messages over possibly untrusted computer networks raises security concerns that need to be addressed. Menezes and others [49] and Schneier [71] list four main information security objectives: First, *confidentiality* ensures that the contents of a message can only be understood by the intended receiver. Second, *data integrity* addresses the unauthorized alteration of data, so that a message cannot be modified in transit. Third, *authentication* denotes the ability of communicating parties to identify each other and prevents others from impersonating a communicating party. And fourth, *non-repudiation* means that an entity cannot deny previous commitments or actions after providing or requesting a service.

The aforementioned security objectives can be achieved using cryptography. The result is a *cryptographic* protocol that uses cryptographic algorithms to ensure one or more of these security properties. Schneier summarizes the purpose of using cryptography in protocols by saying that “it should not be possible to do more or learn more than what is specified in the protocol.”

There are a few building blocks for cryptographic protocols that need to be introduced in brief by describing their basic properties.[3] The first building block are encryption algorithms. *Symmetric encryption* algorithms use the same symmetric key for encrypting a plaintext as for decrypting the ciphertext. Sender and receiver of the message need to agree on the sym-

[3] See Menezes and others [49] or Schneier [71] for an in-depth description of the cryptographic techniques.

metric key before communicating in a secret way. A well-known symmetric key algorithm that is used in this thesis is AES, the Advanced Encryption Standard [54].

In contrast to symmetric key cryptography, *public-key cryptography* uses two different keys for encrypting and decrypting messages, one of them public and the other one private. Anyone with the public key can encrypt messages but not decrypt them. Only the person with the private key can decrypt messages. An example of a public-key algorithm is RSA [69] which is also used in this thesis. Another widely used public-key algorithm is Diffie-Hellman [10] which is used by two communicating parties to agree on a shared key by exchanging messages over a possibly untrusted network. The shared key can then be used to perform symmetric key cryptography.

Two more building blocks of cryptographic protocols are secure hash functions and digital signatures. Schneier defines a *secure hash function*, or one-way hash function, as "a hash function that works in one direction: It is easy to compute a hash value from pre-image [the variable-length input string; the author], but it is hard to generate a pre-image that hashes to a particular value." A common secure hash algorithm is SHA [55] which is also used in this text. *Digital signatures* are used to assure to a recipient that a given message has been created by the claimed sender. One way to implement digital signatures is to invert the use of public and private key of a public-key algorithm: The sender signs a message (or a secure hash of it) with the private key, and other people can verify the signature by using the public key of the sender.

One security property that cannot be solved with cryptography only is *availability*, which is listed by Coulouris and others [7] as a main security objective. Availability denotes protection against an adversary that tries to make a resource or service unavailable. There are different ways how an adversary could make a server unavailable: One way is to flood the service with fake requests so that it becomes too busy to answer legitimate requests, which is called a denial-of-service attack. Another way is to censor an entry in a name system that clients need to resolve the service address in order to request the service.

2.1.3 Properties of Privacy-Enhancing Technologies

The security properties above are widely accepted as protections of both users and providers of services. However, apart from confidentiality, they do not take into account privacy of either service users or providers. The field of privacy-enhancing technologies addresses properties like anonymity or pseudonymity which can be seen as additional requirements to services, exceeding the stated security properties.

Pfitzmann and Hansen [61] have established a terminology for properties of privacy-enhancing technologies, including anonymity and pseudonymity. They assume a basic system model of *senders* sending *messages* to *recipients* using a *communication network*. This model is similar to the system model described above for the definition of distributed systems in general. The authors give a first definition of anonymity: "*Anonymity* of a subject means that the subject is not identifiable within a set of subjects, the *anonymity set*."

The authors further define anonymity in terms of unlinkability of *items of interest* which can be messages or actions such as sending or receiving a message: "*Unlinkability* of two or more items of interest [...] from an attacker's perspective means that within the system [...] the attacker cannot sufficiently distinguish whether these items of interest are related or not." This definition leads the authors to definitions of anonymity for either sender or recipient of a message as well as for the relation between both: "*Sender anonymity* of a subject means that to this potentially sending subject, each message is unlinkable. Correspondingly, *recipient anonymity* of a subject means that to this potentially receiving subject, each message is unlinkable. *Relationship anonymity* of a pair of subjects, the potentially sending subject and the potentially receiving subject, means that to this potentially communicating pair of subjects, each message is unlinkable."

The anonymity definitions can be transferred to the roles of client and server. When considering a client sending a request to a server, the client is referred to as sender and the server as recipient. But obviously, for the response that a server sends to a client, this assignment changes. Therefore, when talking about services, the terms *initiator* and *responder* are used for client and server, as it is done, for example, by Dingledine and others [11]. Hence, *initiator anonymity* comprises both sender anonymity of a client sending messages to a server and recipient anonymity for receiving replies. Likewise, *responder anonymity* denotes recipient anonymity for receiving client requests and sender anonymity for sending replies.

Pfitzmann and Hansen further give definitions for terms related to *pseudonymity*: "A *pseudonym* is an identifier of a subject other than one of the subject's real names. [...] The subject which the pseudonym refers to is the *holder* of the pseudonym. A subject is *pseudonymous* if a pseudonym is used as identifier instead of one of its real names. [...] *Pseudonymity* is the use of pseudonyms as identifiers." The authors further define a sender being pseudonymous as *sender pseudonymity* and a recipient being pseudonymous as *recipient pseudonymity*. These definitions are extended here to *initiator pseudonymity* for a client of a service being pseudonymous and *responder pseudonymity* for a server being pseudonymous. In the context of services, pseudonyms are always *digital pseudonyms*, that is, they are unique as identifiers and suitable for authentication by using them to create digital signatures.

When comparing the two states of a subject being either publicly identifiable by real name or being completely anonymous, pseudonymity covers all states in between. Pseudonymity comprises all degrees of linkability of a pseudonym to a subject. Pfitzmann and Hansen mention two aspects of linkability of pseudonyms: knowledge of the linking between a pseudonym and its holder and linkability due to use of a pseudonym in different contexts.

The knowledge of the linking between a pseudonym and its holder can change over time. Pseudonyms can be initially unlinked, initially non-public, or public from the beginning. The knowledge about a linking can vary from person to person. Unless a pseudonym can be transferred to a new holder (which is excluded by Pfitzmann and Hansen as well as in the discussion here), knowledge of the linking can only increase. Anonymity decreases with increasing knowledge of the linking of a pseudonym to its holder.

The second aspect of linkability covers using a pseudonym in different contexts. Pfitzmann and Hansen distinguish between person, role, relationship, role-relationship, and transaction pseudonyms. A holder using a pseudonym for all transactions uses it as a person pseudonym. Holders may also decide to use a pseudonym for a certain role, like as a company employee or as a private person, or for a relationship to another subject. The holder may also combine both properties and use a distinct pseudonym for a certain role and given relationship, thus using a role-relationship pseudonym. A holder using a new pseudonym for each performed transaction uses a transaction pseudonym, which is closest to anonymity. Using the same pseudonym in different contexts allows establishment of a reputation linked to that pseudonym. But repeated use also reduces the degree of anonymity that a pseudonym can provide.

A special case of linkability due to use of a pseudonym in different contexts can be seen for private services. A server that uses the same pseudonym over time to advertise its service allows others to derive service activity. Whoever can link a pseudonym used by a server to its holder might be able to derive activity of the person providing the service.

2.1.4 Pseudonymous Services

Finally, these definitions suffice to give a definition for pseudonymous services that is used in the following:

> *Pseudonymous services permit clients to request a service from a server using a pseudonym that cannot be linked to the location of the server or the identity of the service provider.*

On the one hand, the pseudonym that is used by a server needs to be persistent, so that clients can request the service using the same pseudonym over time. On the other hand, the linking between the pseudonym and its holder may not be known to anyone but the service provider. The location of the server is explicitly included in the definition, because the linking between the location of a computer, which is usually denoted by its IP address, can be linked easily to a person's identity by the person's Internet Service Provider.

The above definition does not prescribe specific security properties. Typically, pseudonymous services should provide confidentiality, data integrity, and availability. It is also useful to have authentication and non-repudiation of the server. In most cases, clients of a pseudonymous service shall remain anonymous and therefore be able to deny previous requests to the service. Depending on the purpose of a service, clients can use pseudonymous, too, so that they are authenticated to the server and, as a result, cannot deny previous requests.

The next two sections describe privacy-enhancing technologies which either implement pseudonymous services or contain techniques that could be used to do so. A basic distinguishing characteristic of these technologies is whether they transmit messages with high or low latency. High-latency anonymous communication systems are presented in Section 2.2, whereas low-latency systems are discussed in Section 2.3.

2.2 Technologies for High-Latency Recipient Pseudonymity

High-latency anonymous communication systems permit their users to exchange messages in an anonymous or pseudonymous way. Message transmission times typically range from some hours up to one day. While this may be acceptable for asynchronous applications like email, it is insufficient for interactive services like web browsing. Nevertheless, many of the principles behind high-latency anonymity systems have also been applied to low-latency anonymous communication systems. Therefore, a study of high-latency anonymous communication systems is compulsory in order to understand the basic principles behind low-latency systems.

The literature on privacy-enhancing technologies contains a plethora of high-latency anonymous communication systems. While the primary function of these systems is to provide sender anonymity, only a small percentage of them provides sender and/or recipient pseudonymity. The focus here are systems and general principles to provide recipient pseudonymity: the required functionality is that a user Bob can establish a long-term pseudonym to receive messages by a user Alice directed to his pseudonym without anyone being able to link Bob's pseudonym to his real identity.

2.2.1 Usenet Message Pools

An obvious way to achieve recipient pseudonymity is to broadcast a message to all pseudonym holders and let them find out themselves which messages are directed to them and which are not. If all possible recipients have successfully received a message, it is impossible for an external observer to tell to which of them it was directed. If message contents shall be kept confidential, messages can be encrypted for the holder of a recipient pseudonym before broadcasting them. Encryption should not reveal to whom a message is addressed except to the intended recipient. This property is referred to as *implicitly addressing* the broadcasted message to the recipient.

A practical realization of the broadcast idea has been established in 1994 with the Usenet group `alt.anonymous.messages`. If Alice wants to send a message to Bob, she (optionally) encrypts the message and posts it to this newsgroup. Bob periodically downloads all messages and figures out for every message whether he can decrypt them and whether they are directed to him. The only step Bob needs to take to establish his pseudonymous identity is telling it (possibly including an encryption key) to Alice.

On the one hand this approach provides for strong resistance against linking a pseudonym to a recipient's identity. But on the other hand there are obvious scalability problems with this approach.

2.2.2 Pseudonymous Remailers

Pseudonymous remailers take a different, more efficient approach to achieve recipient pseudonymity. The basic idea of a pseudonymous remailer is to act as an intermediary between sender and recipient. A remailer rewrites identifying message headers and forwards the message to the recipient afterwards. A pseudonymous remailer further assigns a pseudonymous identity to every user and keeps a local table containing mappings of pseudonyms to user addresses. Whenever a sender directs a message to a pseudonym, the remailer can look up the recipient's real address and forward the message accordingly. The best known pseudonymous remailer was the Penet remailer `anon.penet.fi` that was set up in 1993 and shut down in 1996. The design of the Penet remailer is unpublished, but a good summary can be found in [60].

A user Bob who wants to create a long-term pseudonym sends an arbitrary initial message to the pseudonymous remailer that assigns a unique pseudonym to Bob like `an144108@anon.penet.fi`. Bob tells his pseudonym to Alice either by sending her a message via the remailer (with the result that his pseudonym is included as sender address instead of his real address) or makes it available to her otherwise. Alice who wants to send a message to Bob simply directs it to his pseudonym, so that it gets routed to the remailer `anon.penet.fi`. The remailer replaces the pseudonymous message recipient with Bob's real address and forwards the message to Bob.

The simple design of pseudonymous remailers implies two major problems: The first is vulnerability to traffic analysis. The remailer does not take precautions to hide the correlation between incoming and outgoing messages. Outgoing messages have similar sizes as incoming messages and are sent at a certain time after the incoming message was received. Further, message contents remain roughly the same (besides removing processing information where to forward the message). An adversary with the ability to monitor traffic could easily link an incoming message directed to a pseudonym to an outgoing message addressed to the pseudonym holder, thus uncovering recipient pseudonymity.

The second major problem is exposure to legal prosecution or hacking attempts. The table containing the mapping between pseudonyms and real addresses is the most sensitive part in the system to ensure recipient (and sender) pseudonymity. Whoever knows this table can uncover pseudonymity of all recipients in the system. Users need to trust the remailer operator in keeping this table secret and protecting it against hacker attacks. Further, it puts the remailer operator at risk of having to disclose the table for legal reasons which in the end was the reason for shutting down `anon.penet.fi`.[4]

[4] The press release announcing the closure of the Penet remailer can be found under: `http://w2.eff.org/Censorship/Foreign_and_local/Finland/960830_penet_closure.announce` (last checked: Dec 17, 2008)

2.2.3 Reply-Block-Based Nymservers

The third approach to achieve high-latency recipient pseudonymity is based on Chaum's *mix-net* design [6]. The basic idea is to relay messages over a chain of remailers with each of them performing cryptographic operations on messages and relaying them in batches. The result is sender anonymity unless an adversary can manage to compromise all remailers in a chain. Chaum also proposed the concept of *untraceable return address* (which will be referred to as *reply blocks* in the following text) to provide recipient anonymity. When combining reply blocks with a *nymserver* (short for pseudonym server), one can further achieve recipient pseudonymity.

The following discussion covers different approaches to achieve recipient pseudonymity using reply blocks. Chaum's mix-net design is described next. While it does not include the description of a nymserver itself, it constitutes the basis for the following approaches. After that, two types of reply-block-based approaches are discussed: The first approach is based on reusable reply blocks in the style of Chaum's untraceable return addresses. The second approach makes use of single-use reply blocks in the attempt to better resist traffic analysis.

Mix Nets and Untraceable Return Addresses

In 1981, David Chaum described the concepts of a *mix* and a *mix net* [6]. The purpose of a mix is similar to that of a remailer: hide the correlation between the sender of a message and its recipient. Only the mix itself would be able to uncover this correlation. A mix net consists of multiple mixes and allows its users to send messages via a cascade of mixes. In this case all mixes would have to collude to link the original sender of a message to the recipient. In addition to relaying messages, a mix performs a couple of operations on relayed messages in order to prevent traffic analysis: received messages are decrypted, padded to a uniform length, reordered, and sent out in regular batches.

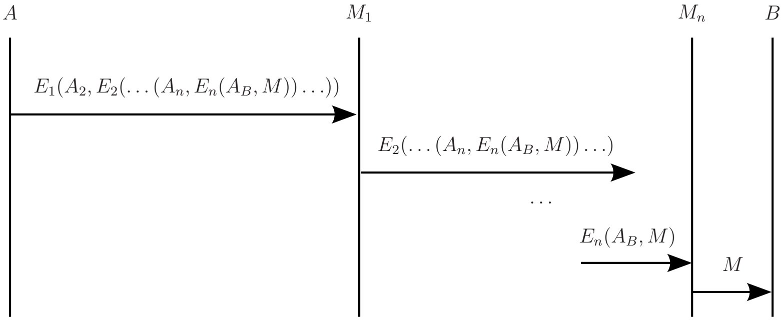

A
M_1
M_n
B
$E_1(A_2, E_2(\ldots(A_n, E_n(A_B, M))\ldots))$
$E_2(\ldots(A_n, E_n(A_B, M))\ldots)$
$\ldots$
$E_n(A_B, M)$
M

Figure 2.1: Sender-anonymous message delivery in a mix net

If user Alice wants to anonymously send a message to user Bob, she first needs to learn about the existing mixes' addresses M_i and their public keys PK_i as well as Bob's address A_B. Alice prepares her message for Bob by adding Bob's address and encrypting her message for the last remailer in the chain M_n. Next she adds the address of the last remailer A_n and encrypts the result for the last but one mix M_{n-1}. She subsequently adds similar layers for the other mixes in reverse order from M_{n-2} to M_1. Finally, she can send the composed message to M_1. The mixes M_i all perform the same task of decrypting the received message and forwarding the result to either the next mix in the chain or to Bob, respectively. Figure 2.1 shows the exchanged messages of Alice anonymously sending a message to Bob.

Chaum also proposed a similar technique for anonymously sending a reply message back to the sender. An *untraceable return address* is constructed by the sender and made known to another user, possibly together with a sender-anonymous message. The recipient then can reply to the sender without knowing the sender's real address. An untraceable return address only contains the routing portion of a message while the message content M is added later by the user who actually sends the reply. Untraceable return addresses contain symmetric keys R_i for all mixes on

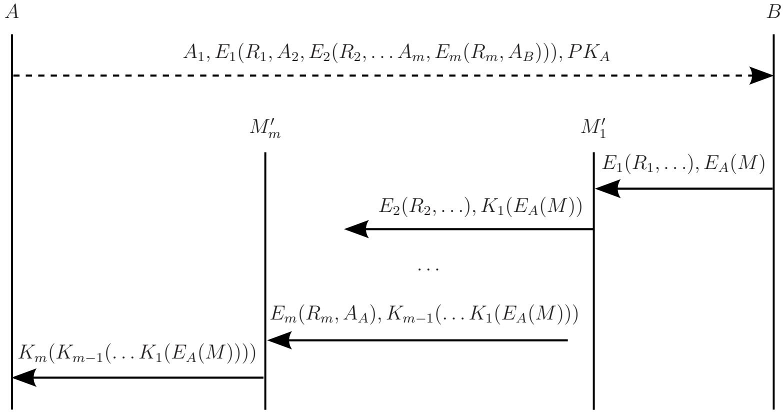

A
B
$A_1, E_1(R_1, A_2, E_2(R_2, \ldots A_m, E_m(R_m, A_B))), PK_A$
M'_m
M'_1
$E_1(R_1, \ldots), E_A(M)$
$E_2(R_2, \ldots), K_1(E_A(M))$
$\ldots$
$E_m(R_m, A_A), K_{m-1}(\ldots K_1(E_A(M)))$
$K_m(K_{m-1}(\ldots K_1(E_A(M))))$

Figure 2.2: Sender-anonymous reply delivery using an untraceable return address

the path that are used to encrypt the reply, which differs from decrypting forward messages using the private keys of the mixes. Figure 2.2 depicts the sequence of exchanged messages that are necessary for Bob to reply to Alice using a reply block.

Chaum's design obeys an important limitation: a mix may not process the same message twice. If this operation would be permitted, an adversary could re-insert a message to a mix and find the next mix or the recipient's address in the intersection of both outgoing batches. Mixes ensure replay protection by memorizing forwarded messages and dropping duplicates. As a result, untraceable return addresses can only be used once.

Cypherpunk-Style Nymservers

Cypherpunk remailers, as described by Goldberg in [24], were designed to overcome the weaknesses of pseudonymous remailers as described in Section 2.2.2 by applying (most of) the principles of Chaum's mix nets. Remailers shall not store any sensitive data about users or relayed messages that could be exploited by an attacker or required to be disclosed due to legal pressure.

The basic function of a Cypherpunk remailer remains the one of stripping headers from received messages and forwarding them afterwards. In addition to that, Cypherpunk remailers recognize (but do not enforce) a couple of commands while processing a message: Messages can be encrypted using the public key of a remailer, so that the remailer needs to decrypt the message before further processing it. The sender of a message can further specify a random or fixed time for which a message shall be delayed before being forwarded. Further commands like these have been introduced over time, but not all remailers are required to support them.

Besides sending forward-anonymous messages, Cypherpunk remailers also support reply blocks to enable replies to anonymous messages. A major difference to the original design of Chaum's untraceable return addresses is that Cypherpunk remailers permit using reply blocks multiple times.

Cypherpunk-style nymservers like `nym.alias.net` [47] further provide recipient (and sender) pseudonymity. The idea is to combine the concepts of a nymserver with reply blocks instead of real addresses. The nymserver stores a table between pseudonyms and one or more reply blocks that can be used to deliver a message to the pseudonym holder.

A user Bob who wants to create a long-term pseudonym first creates an asymmetric key and a reply block directed to his real address.[5] He then deposits the public key, the reply block, and a chosen pseudonymous identifier at the *nymserver*, using a remailer chain himself to hide his identity from the nymserver. Finally, Bob announces his pseudonym to other users who might want to contact him.

If Alice wants to send a message to Bob, she addresses her message to

[5] Alternatively, he can combine usage of reply blocks with the approach to direct messages to a usegroup message pool as described in Section 2.2.1 and insert the address of a Usenet group instead. Even if an attacker would reveal the destination of a reply block, only the Usenet group would be revealed.

Bob's pseudonym at the nymserver, for example `nymB@nym.alias.net`. The nymserver first ensures that the message is not a replay of a previous message; if it is, the server drops the message. Otherwise, the nymserver encrypts Alice's message with Bob's public key and forwards it together with the stored reply block to the first remailer listed in Bob's reply block. The message is then delivered via the remailer chain like a usual reply message. Upon receiving the message from the last remailer, Bob decrypts it successively using all symmetric keys that he included in the reply block and finally with the private key of his pseudonym.

As an extension of the basic case described above, a user could also deposit more than one reply block at the nymserver and define a rule whether messages shall be sent using all or only a random subset of these reply blocks. The additional paths can be used either to improve reliability or to create cover traffic by creating reply blocks with long remailer chains ending nowhere.

Cypherpunk-style remailers exhibit a couple of security problems as described by Lance Cottrell.[6] Cypherpunk remailers do not take reasonable precautions to prevent an adversary from associating incoming with outgoing messages. Message sizes are not unified and decrease after each processing step due to either dropping an encryption layer or a reply block layer. An adversary can further guess message correlations from the timing in which messages are received and forwarded to the next remailer. The fact that reply blocks may be used repeatedly makes it possible for an adversary to trace the path to a recipient by sending a large number of messages.

Mixmaster remailers [50] solve most of these problems by adding message padding and other features described by Chaum [6], including strict prevention of replayed messages. Mixmaster remailers further perform

[6] See the essay by Lance Cottrell on Mixmaster and remailer attacks: `http://www.obscura.com/âĹijloki/remailer/remailer-essay.html` (last checked: Dec 17, 2008)

integrity checks of all messages to make sure that they have not been modified. This integrity check, however, makes it impossible to construct headers of reply messages without already knowing their content which is added by the recipient. Consequently, the Mixmaster protocol does not support recipient pseudonymity.

The Babel [27] design contains a slight but noteworthy modification of reply blocks by including a *key seed* in reply blocks that is encrypted for the creator of the reply block. This key seed is used to derive all symmetric keys for the return path. The advantage is that the sender does not have to remember the keys of a reply block in order to process a reply, but can reconstruct them using the enclosed key seed. This modification of Cypherpunk-style reply blocks allows the sender to remain stateless with respect to outstanding replies.

Mixminion-Style Nymservers

The Mixminion system [8, 46] changes the design of reply blocks by restricting them to be used only a single time. *Single-use reply blocks* are not exposed to replay attacks, because Mixminion remailers detect replays for both forward and reply messages. Another feature of reply blocks in Mixminion is that reply messages that are sent using a reply block are indistinguishable from usual forward messages, even to the forwarding remailer. Since replies are rare in number compared to forward messages, this makes it even harder for an adversary to trace a reply message.

Accordingly, Mixminion-style nymservers are designed to handle single-use reply blocks for a given pseudonym. The Mixminion designers [8] proposes two approaches for nymservers based on single-use reply blocks. The first approach assumes that a pseudonymous user deposits a sufficient number of reply blocks on a nymserver, so that all incoming messages can be delivered. Whenever the number of reply blocks stored at the nymserver decreases, the user would have to upload new ones to keep his pseudonym operational. The main disadvantage of this approach is that an attacker can deny service by flooding any pseudonym to consume all stored reply blocks and make further message delivery impossible.

The second nymserver approach in Mixminion resembles a mailbox design. The nymserver stores all received messages in the first instance (possibly encrypting them upon reception to lower the risk of disclosure). A user periodically queries the mailbox for newly received messages and supplies an appropriate number of reply blocks, so that the nymserver can deliver the messages. A disadvantage of this approach is that the nymserver needs to store user messages potentially for a long time, which might have legal and security implications.

2.2.4 Private Information Retrieval

The Pynchon Gate [70] constitutes a rather different approach to recipient pseudonymity. Instead of delivering messages to recipients over a remailer network that supports reply messages, the Pynchon Gate makes use of private information retrieval techniques. The system processes received messages in daily batches and makes them available in an encrypted and specifically indexed way to all possible recipients. Clients request a subset of all messages following a schema that does not permit an observer to determine the pseudonym the recipient is using and significantly reduces the overhead as compared to downloading all messages.

The Pynchon Gate system consists of a central component that processes incoming messages with the two subcomponents of a *nymserver* and a *collator*. The nymserver encrypts incoming messages for the recipients as soon as they arrive and passes a batch of all messages to the collator once every day. The collator packages them into an indexed bucket pool and replicates the index and all messages to a set of independently operated *distributor nodes*. The pseudonym holders first download the complete index to calculate which buckets hold their messages and then re-

Table 2.1: Comparison of high-latency designs to achieve recipient pseudonymity

	Usenet	Penet	Cypher-punk	Mix-minion	PIR
No single point of trust	⊕	⊖	⊕	⊕	⊕
Traffic analysis resistance	⊕	⊖	⊖	⊕	⊕
Usability for pseudonym holder	⊖	⊕	⊖	⊖	⊖
Usability for sender	⊖	⊕	⊕	⊕	⊕

trieve their buckets using private information retrieval. Therefore, they send k requests to different distributor nodes for each bucket they want to retrieve in a way that the bucket content can be obtained by combining all k results. However, an observer that sees only $k - 1$ requests cannot derive the bucket that a nym owner was asking for. Users always request the same number of buckets to conceal the number of messages they actually receive.

One disadvantage of this approach is that pseudonym holders need to download a volume of messages of the maximum receivable size every cycle, regardless of the fact whether they receive any message at all. Users need to estimate the volume of messages they typically receive in advance.

2.2.5 Comparison

Despite their different approaches, all presented designs can be used to achieve high-latency recipient pseudonymity. However, the approaches differ in certain criteria like security properties, efficiency, and usability. The approaches are compared with respect to these criteria in Table 2.1.

The first criterion is having *no single point of trust*. Systems that depend on one or a few single points of trust for meeting security properties like pseudonymity are vulnerable to threats like legal pressure and hacking attempts. Only the Penet pseudonymous remailer is attackable in this regard which finally led to termination of its service. In the other systems there is no single point that knows about the link between a pseudonym and the real identity of its holder, thus providing better protection.

The second criterion is *resistance against traffic analysis*. An adversary who can keep track of a message in a system by comparing message sizes and timings might be able to link a pseudonym to its holder. The Penet pseudonymous remailer does not take special precautions to impede traffic analysis and is thereby vulnerable to traffic analysis. The Cypherpunk pseudonymous remailer introduces a few countermeasures against traffic analysis, but does not enforce them. Further, the fact that reply blocks may be used multiple times defeats this protection. Mixminion implements effective countermeasures against traffic analysis including indistinguishability of forward and reply messages. Usenet message pools and Private Information Retrieval make traffic analysis attacks very hard, because messages addressed to a recipient are hidden in larger message sets that a recipient downloads regularly from the system.

Usability of pseudonym holders is another criterion when comparing the approaches. Pseudonym holders in the Usenet-based approach and in the Private Information Retrieval system need to periodically download a possibly large number of messages in order to determine whether any of them is addressed to them or not. Mixminion requires pseudonym holders to provide enough single-use reply blocks to receive messages addressed to them which requires special software. Establishing a recipient pseudonym in the Cypherpunk remailer either requires a couple of manual steps or special software, too. Only the Penet makes registration of a new recipient pseudonym a simple task that only requires sending a single mail to the nymserver. Incoming messages for the pseudonym are forwarded to the real address of the recipient.

Usability is also a factor when considering the *sender* of a message. In the Usenet approach users need to prepare messages by encrypting them for pseudonymous recipients, possibly using special software. In the other approaches this is not necessary, so that messages can simply be addressed to special pseudonyms and are delivered by the pseudonymous communication system to the intended recipients.

2.3 Technologies for Low-Latency Responder Pseudonymity

A characteristic feature of most high-latency anonymous communication systems is the introduction of artificial delays to defeat traffic analysis. In contrast to this, low-latency anonymous communication systems do not introduce any delays artificially, but focus on fast transmission of messages. As a result, interactive protocols can be executed on top of the anonymous communication system. As with high-latency systems, there are different designs to provide anonymity or pseudonymity in a low-latency system. The following description focuses on approaches that support responder pseudonymity (rather than only recipient pseudonymity) with a long-term pseudonym which is a necessary prerequisite to implement pseudonymous services.

2.3.1 ISDN Mixes

Pfitzmann and others [62] presented in 1991 the first design for a low-latency anonymous communication system supporting responder pseudonymity. The purpose was to provide untraceable communication in the digital telephony network ISDN with the bandwidth restriction of using only two duplex data channels and one signalling channel.

The main goal of the approach was to create a *mix channel* transmitting a continuous stream of data from initiator to responder with almost no delay and without anyone being able to trace either initiator or responder. The basic building block to transmit data is an *ISDN mix* that is a variant of Chaum's mix [6], but that processes data in real-time. Therefore, the initiator prepares a message for a mix cascade M_1 to M_n by encrypting it multiple times to the public keys of all mixes in reverse order. Every layer of encryption contains a symmetric *decryption key* k_i that is used by the corresponding mix M_i to decrypt succeeding stream data. After establishing such a *mix sending channel,* the initiator can send a data stream which is encrypted multiple times to the first mix which is then decrypted by every mix using the appropriate secret key k_i.

Likewise, participants create such channels to receive stream data instead of sending it. Therefore, a participant sends an establishment message to a series of mixes containing symmetric *encryption keys* k_i'. In this case unencrypted stream data is provided to the last mix M_n and encrypted by every mix in the cascade until it is delivered to the participant. This concept of a *mix receiving channel* bears resemblance to untraceable return addresses as presented in Section 2.2.3, however, with the difference that participants establish receiving channels themselves instead of giving out information for how to establish them to others.

So far it is only possible to create sending and receiving channels. What is still missing is a way to connect two channels to transfer user data from an initiator to a responder. Therefore, two participants agree on a common label that they include in their mix-channel establishment messages and that is processed by the last mix M_n to connect the two channels. The initiator creates such a label and broadcasts it to all possible responders using a sending channel.[7] The broadcasted message is implicitly addressed to the responder, so that the responder can cryptographically determine that the message is addressed to him and read the label, while the message is incomprehensible for other participants. Initiator and responder of this broadcast message use the label to create a sending and receiving mix channel, respectively, and the last mix M_n connects them.

Even though some ideas of this approach could be applied to Internet services, there is a major assumption that is specific to circuit-switched

[7] The approach provides for a hierarchical network, so that these messages would in fact be broadcasted to a certain number of participants only.

telephony networks: users have a fixed amount of bandwidth at hand. On the one hand, this bandwidth can never be exceeded, so that resources need to be released as quickly as possible. But on the other hand, unused bandwidth can be used for continuously sending cover traffic which is often too expensive in packet-switched networks like the Internet. Further, the requirement to send and receive broadcast messages is unrealistic on an Internet scale, even when being performed in a hierarchical fashion.

2.3.2 Onion Routing

The first Onion Routing design [25, 67] was proposed in 1996 and provides anonymity for the communication partners of Internet services like the World Wide Web and Telnet. Onion routing is based on a possibly large number of *routing nodes*. In order to establish an anonymous connection, the initiator selects a series of routing nodes and creates an *onion* which encapsulates that route. The onion contains encrypted layers for the routing nodes in the path, including secret decryption/encryption keys for each routing node. The initiator sends the onion along the selected route, thus establishing a *virtual circuit* beginning at the initiator and ending at the last routing node. The routing nodes store the state of a virtual circuit and process data going in either forward or backward direction by decrypting/encrypting it and forwarding it to either the next or previous routing node in the path. The routing nodes further make sure that onions are not used more than once by checking their timestamp to see if they are still valid and by memorizing processed onions until they expire.

The design also introduces the concept of *reply onions* which can be used by any participant to create a virtual circuit to the creator of the reply onion. The idea of reply onions resembles the concept of untraceable return addresses in as much as they permit a responder to create a virtual circuit back to an anonymous initiator. Just like normal onions, reply onions contain a pre-defined path of router nodes and include all routing information and cryptographic keys, encrypted in layers, that is necessary to build the virtual circuit. Reply onions may only be used once, too, which is ensured by the routing nodes.

Reply onions can be used in different ways to achieve responder pseudonymity. The typical way is to include a reply onion in the data stream of an initiator-anonymous connection, so that the responder can contact the initiator after the virtual circuit has been torn down. A participant could also broadcast reply onions which could then be used anonymously by other participants to establish a virtual circuit to the first participant. In addition to that, the authors [25] briefly mention two more approaches to create a completely anonymous connection between two parties that are based on the concept of an *anonymity server*. In the first approach two participants create virtual circuits to the same anonymity server that connects the two circuits using a shared token. In the second approach one participant creates a virtual circuit to an anonymity server and requests it to create a connection to another participant using a provided reply onion. The results of both approaches would be that the identities of both participants are protected by a virtual circuit that they have determined themselves.

Unfortunately, the usefulness of reply onions to implement long-term responder pseudonyms is limited. First, reply onions can only be used once by design. Second, validity of reply onions needs to be restricted in order to limit the amount of storage that is necessary to memorize previously processed reply onions. Third, using a reply onion requires all routing nodes that were picked during the creation process to be still available at the time of building the virtual circuit. Nevertheless, reply onions constitute an important step in the development of responder pseudonymity.

2.3.3 TAZ Servers and Rewebber Network

TAZ servers[8] and the Rewebber Network [23] were proposed in 1998 by Goldberg and Wagner as a means for anonymous publication on the World Wide Web. *Rewebbers* are HTTP proxies that understand so-called *rewebber locators* that basically resemble the concept of untraceable return addresses: rewebber locators consist of a rewebber address and an encrypted string possibly containing another nested rewebber locator. Clients use rewebber locators to request documents from servers that wish to remain anonymous by sending them to the first rewebber. After decrypting the encrypted URL part, the rewebber forwards the locator to the next rewebber until it finally reaches the real web server which originally created the rewebber locator. The webserver responds with a document which is encrypted multiple times and passed back to the client. Each rewebber on the way removes one layer of encryption, so that the client receives the plain document as response to the original request. The result of this approach is that only the rewebber closest to the client sees decrypted data whereas only the rewebber closest to the server learns where these data originate from.

The proposed design further contains the concept of *TAZ servers* to achieve pseudonymous publication on the World Wide Web using short persistent names. TAZ servers store mappings of persistent domain names in the virtual .taz domain to rewebber locators. This way clients do not need to remember the rather cumbersome rewebber locators, but a much shorter .taz domain name. Further, if a rewebber locator needs to be replaced, the server does not need to inform all clients, but only update the mapping at the TAZ server. As a safeguard to impersonation attacks on existing .taz servers, the TAZ server may store a password hash to authenticate updates. TAZ servers do not, however, provide authenticity of

[8] TAZ stands for *Temporary Autonomous Zone* which is inspired from the book *T.A.Z.: The Temporary Autonomous Zone, Ontological Anarchy, Poetic Terrorism* by Hakim Bey.

stored entries with regard to the person or organization running the webserver. The authors state that clients and servers should rely on end-to-end authentication by signing all anonymous documents using a private key and distributing the public key together with the .taz domain name.

The main problem of this approach is that rewebber locators can be used multiple times; while caching at the rewebbers reduces the number of requests that need to traverse the complete path, the general problem of traceability remains. Another minor weakness of the design of TAZ servers is missing authenticity of .taz domain names: a client that requests a resource from a trusted .taz service and obtains a rewebber locator for that resource from a TAZ server needs to download the resource first before being able to authenticate its origin; it would be desirable to perform authentication before actually performing the request.

2.3.4 Pseudonymous IP Network

The Pseudonymous IP (PIP) Network [19] as proposed by Goldberg in 2000 provides anonymity and pseudonymity for clients and pseudonymity for servers of Internet services. The central concept to achieve anonymity for clients is the *IP wormhole*. An IP wormhole enables a client to anonymously exchange IP packets with an *Anonymous Internet Proxy* which is part of the PIP network. Roughly speaking, IP wormholes work similarly to virtual circuits in the Onion Routing [25] design, but on the IP level rather than on the TCP level; the differences between the two concepts do not affect the way of providing pseudonymity for servers as discussed here.

The design contains the concept of a *rendezvous server* that can, but does not need to be part of the PIP network. A service provider registers his service using an IP wormhole at a rendezvous service under an arbitrary *service tag*; while the only requirement to the service tag is that it is unique, it may also be the hash of a public key that is used by the rendezvous server to authenticate the message as originating from the owner of the private key. In consequence, the rendezvous server assigns one of its public IP addresses (assuming that it controls multiple IP addresses; otherwise it assigns its own IP address) and possibly a TCP/UDP port number to the service and waits for TCP connection requests, UDP datagrams, or simply IP packets. The rendezvous server further publishes the service tag together with the assigned IP address and port number to a distributed storage network like Gnutella [18].

A client looks up a given service tag in the distributed storage network to retrieve IP address and port number of the rendezvous server. Depending on the privacy requirements, the client can either contact the rendezvous server directly or use an IP wormhole itself to protect its real IP address. All data between client and server are hereafter forwarded by the rendezvous server in both directions. If confidentiality is needed, client and server need to apply end-to-end encryption, because otherwise data is readable for the rendezvous server.

Rendezvous servers might have transient nature and go offline during the lifetime of a service. In order to stay connected, services should set up connections to multiple rendezvous servers and advertise them under the same service tag in the distributed storage network. The author further sketches an extension to switch rendezvous servers seamlessly during an ongoing connection between client and server.

One problem of the approach is that the distributed storage network exhibits only weak security properties: an adversary could publish an arbitrary number of entries for the same service tag, making it hard for clients to identify the legitimate ones. The author discusses the possibility to let entries be signed by the service provider and verified by the clients. However, this requires a change in the design, as entries are generated and published by the rendezvous server and not by the service provider; it also puts the burden of downloading and verifying a possibly large number of entries to clients. It would be better to implement authentication at the distributed storage nodes rather than at clients and servers.

2.3.5 Tarzan

In 2002, Freedman and Morris presented Tarzan [16, 17], a peer-to-peer anonymous IP network overlay. In this system every participant acts as a proxy and relays data for other participants. In combination with cover traffic an adversary cannot tell easily whether traffic originates at a certain proxy or is relayed for another participant. The network meta-data is stored in a distributed fashion, so that there is no central database in the network.

Participants can build *tunnels* to other proxies which are based on layered encryption and multi-hop routing as in the previously described approaches. The last proxy of a tunnel acts as *network address translator* to bridge traffic between the Tarzan network and the Internet. This address translation works in both directions, so that a service provider can establish a tunnel and publicize the address of the network address translator to others as a way to contact him. Clients can then establish a connection to the network address translator which forwards requests to the server and delivers responses back to the client.

The major problem of this approach is that the Tarzan design does not address the effect of volatility of proxies to service availability. Once a proxy that is part of a tunnel leaves the network, the server needs to establish a new tunnel to the network address translator and is unavailable in the meantime. Even worse, if the network address translator disappears, the server needs to advertise a new service address for its service. In short, the design is missing a naming service for pseudonymous services.

2.3.6 I2P

I2P, the Invisible Internet Project, is another peer-to-peer-based approach to achieve responder pseudonymity. The design is unpublished, so that the following description is based on the information given on the project website.[9] Every participant of the I2P network runs a router that may initiate connections to other routers itself and relay traffic for other routers. Participants build uni-directional *tunnels* through previously selected paths of routers to either send IP packets to the router at the end of the tunnel (outbound tunnels) or receive data from there (inbound tunnels).

The default operation of I2P, as opposed to the previously described designs, is to establish a connection to another, pseudonymously identified I2P node while connections to non-I2P nodes explicitly require an outbound proxy running on an I2P node. I2P nodes store so-called *lease sets* in the distributed *network database*. A lease set contains a participant's public key, a set of inbound tunnels, and a signature created with the participant's private key. Tunnels are valid for only 10 minutes, so that a participant needs to refresh his lease set that often. A client can request a lease set of another node by querying the database for the hash of the participant's public key and connect one of her outbound tunnels to one of the retrieved inbound tunnels. As a means to counter attacks on specific parts of the distributed database which could make a lease set unavailable, entries are stored under daily changing keys by using the hash of the concatenation of the current date and the hash of the participant's public key as storage key.

The design of responder pseudonymity in I2P is promising, but without any published information besides the website it is hard to evaluate security properties of the system.

2.3.7 Tor

Tor [11] evolves the original Onion Routing design [25]. Tor contains a feature to provide responder pseudonymity that is called *location-hidden*

[9] See also the I2P homepage: `http://www.i2p2.de/techintro.html` (last checked: Dec 17, 2008)

services, or *hidden services* in short. The design of hidden services refines Goldberg's rendezvous design [19] by connecting two circuits created by client and server on a common *rendezvous point*. Hidden services further use separate *introduction points* as first contact points for clients to improve denial-of-service protection and a *hidden service directory* system to store hidden service descriptors.

A service provider who wants to offer a hidden service generates a public key pair as long-term identity for his service. The server establishes the hidden service in the Tor network on randomly selected relays that act as *introduction points*. In order to do so the server creates *circuits* consisting of three relays and establishes introduction points on the last relays of these circuits. The server sends establishment messages to these relays that are created with the private key of the hidden service. The first two relays in the circuit protect the link between the hidden service identity that is known to the introduction point and the IP address that is known to the first relay in the circuit. In the next step the server creates a *hidden service descriptor* containing the list of previously established introduction points, signed with the private key of the service. The server uploads this descriptor to the hidden service directory using a circuit to hide the link between its address and the hidden service identity from the hidden service directory. Finally, the server advertises the hash of the hidden service's public key, the *onion address*, to prospective clients. After that, the hidden service is established and ready to be contacted by clients.

A client that wants to establish a connection to a hidden service fetches its hidden service descriptor from the hidden service directory. The client uses a circuit for this request to hide its attempt to access a certain hidden service from the hidden service directory. If a descriptor is available, the client establishes a separate *rendezvous point* in the same way as the server established introduction points. The rendezvous point will be used to transfer application data. The client further opens a circuit to one of the service's introduction points. As soon as the rendezvous point is established, the client sends an introduction request to the introduction point that is forwarded to the hidden service. The server learns about the rendezvous point of the client and opens a circuit to that relay. The server then sends a rendezvous message to the rendezvous point that is passed on to the client. At this point, client and server share a common circuit. The client can then attach application streams to that circuit and send requests to the hidden service.

The hidden service design exhibits a couple of useful security properties. Hidden services can resist certain attacks by quickly changing their introduction points and publishing a new descriptor containing new ones. The service is authenticated towards the introduction points, the hidden service directory, and clients, so that nobody can impersonate an existing hidden service. However, there are also a few drawbacks of the hidden service design. Clients require special software, the Tor software, to be able to access a hidden service. Further, the hidden service directory, consisting of three fixed servers, constitutes single points that are able to censor any hidden service.

The hidden service design is the most evolved pseudonymous services design nowadays. It is deployed in a network consisting of more than 1,000 relays and a few hundred thousands users. The Tor system, including the hidden service protocol, is well-documented and backed up by an active community. These facts make Tor hidden services the most promising candidate for implementing private services as motivated in the previous chapter.

2.3.8 Comparison

All described designs support responder pseudonymity by allowing clients to establish a connection to a server under a given pseudonym without knowing its real location. The approaches shall be compared using a number of criteria ranging from security to usability properties. The ISDN-

Table 2.2: Comparison of low-latency designs supporting responder pseudonymity

	Onion Routing	TAZ/Re-webber	PIP	Tarzan	I2P	Tor
Long-term pseudonyms	⊖	⊕	⊕	⊖	⊕	⊕
Responder authentication	⊖	⊖	⊖	⊖	⊕	⊕
No single point of censorship	⊕	⊖	⊕	⊕	⊕	⊖
Traffic analysis resistance	⊕	⊖	⊕	⊕	⊕	⊕
Performance	⊖	⊖	⊕	⊕	⊖	⊖
Usability for initiator	⊖	⊕	⊕	⊕	⊖	⊖

based design is excluded from this comparison, because its assumptions are too different from the other approaches that are designed for packet-switched networks like the Internet. Table 2.2 shows an evaluation of the approaches with respect to the discussed criteria.

The first criterion is the ability of a responder to *maintain a long-term pseudonym*. This ability is a prerequisite for offering a pseudonymous service that can be contacted by clients using the same pseudonym. The original onion routing does not exhibit this feature but only sketches briefly how clients can learn about and use reply onions. The Tarzan system binds a responder pseudonym to a Tarzan node in the system that might vanish at any time, leaving no way of contacting the responder anymore. The other presented designs permit responders to maintain a long-term pseudonym by using a directory to map it to short- or medium-term contact information.

From the designs that support long-term pseudonyms, only a subset ensures *authenticity of the responder* using a pseudonym. These designs include I2P and Tor hidden services where the mapping between long-term pseudonym and short- or medium-term contact information is signed by the responder. Neither the TAZ/Rewebber system nor the PIP system exhibit authentication as a mandatory feature. Onion Routing and the Tarzan system do not provide authentication of responder pseudonyms as they lack support for long-term pseudonyms in the first place.

In the context of how long-term pseudonyms are implemented, another criterion to compare the approaches is censorship resistance. If the mapping between long-term pseudonym and short- or medium-term contact information is stored at a single point, it could be censored to make a responder unavailable. This censoring might be possible in the TAZ/ Rewebber system as well as in Tor hidden services. The PIP design and I2P use distributed approaches to store mappings for long-term pseudonyms which are significantly harder to censor. Onion Routing and Tarzan do not have a single point of censorship as they do not support long-term pseudonyms.

Further, the approaches shall be compared with respect to their *traffic analysis resistance*. None of the approaches introduce delays on purpose to defeat traffic analysis of a global passive adversary who can observe the whole network. But traffic analysis can also be performed by creating new paths through a network beginning or ending at the victim and observing recurring traffic patterns. The TAZ/Rewebber system is vulnerable to this attack, because a rewebber locator may be reused for an arbitrary number of requests. Tor hidden services have been proven vulnerable to a similar attack where an adversary could force the server to create new circuits, possibly including a rogue node at the first position; however, this vulnerability has been fixed [56]. The other systems are not vulnerable to this attack, because paths are established by the responder, as in PIP, Tarzan, or I2P, or can only be established once, as in the original Onion Routing design.

The *performance of establishing a connection* is another important criterion with regard to usability. The original Onion Routing design, the TAZ/Rewebber system, I2P, and Tor hidden services require the initiator to create a new path or part of it through the network for every connection. On-demand path creation slows down connection establishment to a pseudonymous responder as compared to approaches where the initiator only needs to connect to a publicly accessible node in the network, as in the PIP system and Tarzan.

Finally, *usability for the initiator* is different when comparing the approaches. The TAZ/Rewebber system, the Pseudonymous IP network, and Tarzan allow the initiators of connections to pseudonymous responders to do so using standard software, like a Web browser. In contrast to this, Onion Routing, I2P, and Tor require the initiators to use special software when contacting a pseudonymous responder. This requirement reduces usability to a certain extent.

This chapter has given a definition of pseudonymous services and has presented the necessary background on technologies to realize them. As a result, Tor hidden services [11] have been identified as a promising candidate to implement private services which use pseudonymous services as a basic building block. The next chapter will give more specific background on Tor hidden services which is required in order to understand the missing pieces for implementing private services.

3 Tor Hidden Services

Tor hidden services permit users to provide a service to other users without leaking the location of the server. Concealing the server location is the first step in hiding the user's activity and protecting the server from attacks. In this chapter the Tor system, including the hidden service feature, is described in more detail in order to give enough background for the subsequent contribution chapters. The description starts with two general features of Tor, namely circuit creation in Section 3.1 and the directory system in Section 3.2. The hidden service protocol is presented in more detail in Section 3.3. Finally, Section 3.4 describes the threat model of Tor which defines the capabilities of an adversary that the system is able to protect against.

3.1 Circuit Creation

The Tor network is an overlay network consisting of *relays* that transport user data and *clients* that operate on behalf of users. Clients and relays exchange fixed-size *cells* over TLS-encrypted *connections* [9] to build multihop *circuits* and attach *streams* to them. In contrast to the original onion routing design, circuit creation in Tor exhibits perfect forward secrecy by negotiating ephemeral session keys rather than using long-lived public keys. This process is called *telescoping* and is performed incrementally. Figure 3.1 shows the sequence of circuit creation and stream attachment that is described in the following.

A client starts to create a circuit by selecting (typically three) relays based

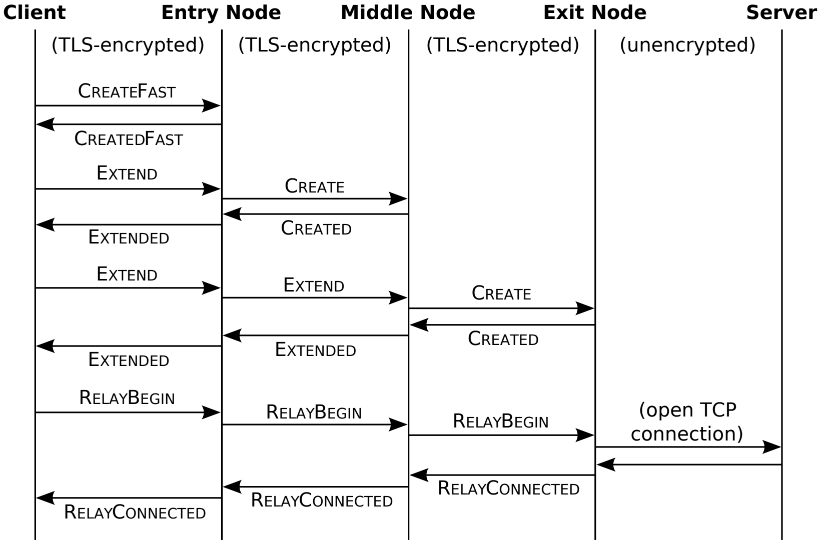


Figure 3.1: Circuit creation and stream attachment

on criteria like bandwidth, availability, or policy to exit to a certain target.[10] The three relays are also referred to as *entry*, *middle*, and *exit node* depending on their position in a circuit. The client establishes a connection to the first relay using TLS which provides authentication of the relay towards the client and confidentiality of communication. The client then sends a CreateFast cell with the first half of a secret key to encrypt data in the newly established circuit. The relay responds with a CreatedFast cell with the second half of the secret key, after which both client and relay share a secret key. It is not necessary to perform a Diffie-Hellman key exchange at this point, because the client has already authenticated the relay and both are communicating over an encrypted connection. This is not the case for the subsequent nodes in the circuit to which a client does not

[10] It is assumed here that clients know the list of all relays, their configuration, and public keys which will be described in more detail below.

open a direct TLS connection.

In the next step the client extends the one-hop circuit incrementally to the other relays. Therefore, the client sends an EXTEND cell to the first relay that is encrypted with the previously negotiated secret key[11] and that contains a nested CREATE cell that is encrypted for the public key of the second relay. The first relay decrypts the EXTEND cell, establishes a TLS connection to the second relay, and forwards the encrypted CREATE cell. The CREATE cell contains the first half of a Diffie-Hellman key handshake [10] to establish a secret key between client and second relay. The second relay performs the Diffie-Hellman handshake and responds to the first relay with a CREATED cell containing the second half of the Diffie-Hellman handshake. The first relay encrypts the received CREATED cell with the secret key, encapsulates it in an EXTENDED cell and sends it to the client. Subsequent circuit extensions work likewise with all cells except CREATE and CREATED being encrypted multiple times using the previously negotiated secret keys.

As soon as a circuit is established, the client can attach one or more application-level streams to it. Therefore, the client prepares a RELAYBEGIN cell by encrypting it to the secret keys and sending it to the first relay in the circuit. Every relay removes the outer encryption layer and forwards the cell to the next relay. The last relay finds the RELAYBEGIN cell and opens a connection to the given target. After the connection has been established, the last relay creates a RELAYCONNECTED cell, encrypts it with the shared secret key and sends it to the previous relay in the circuit. Again, every relay encrypts the received cell and forwards it to either the previous relay in the circuit or to the client. Finally, the client decrypts the cell with all secret keys and learns that the stream has been opened. Subsequent data is sent similarly in both directions contained in RELAY-

[11] To be precise, client and relay use keys that are derived from the negotiated secret key to encrypt data that is sent in either forward or backward direction; however, this level of detail is not required for the discussion here.

DATA cells. Both sides can close the stream by sending a RELAYEND. After all streams have been closed and the circuit reached a certain age, the circuit can be closed by sending a DESTROY cell in either direction. A more in-depth specification of circuit creation in Tor can be found in the Tor specification [64].

The circuit creation process in Tor may appear rather cumbersome. However, its purpose is to achieve *perfect forward secrecy*. In the original Onion Routing design [25] a single onion is used to establish a virtual circuit that is encrypted using medium-term public keys. A hostile node could record traffic and later force successive nodes to decrypt it. This threat does not emerge with the telescoping approach in Tor. Once session keys are deleted, nobody can force or compromise a relay to decrypt old traffic. The circuit creation algorithm further provides authentication of relays to the initiating client so that an adversary cannot easily impersonate relays which was shown by Goldberg [21].

Building circuits in Tor can be a time-consuming task and has therefore been subject to various investigations. Kate and others [32] proposed an alternative circuit building protocol that requires only a single pass and also provides forward secrecy. Øverlier and Syverson [58] presented a modified protocol based on Diffie-Hellman handshakes to build Tor circuits with fewer exponentiations than in the original protocol thus accelerating the process. Panchenko and others [59] measure the influence of single overloaded nodes on the general performance of Tor. They propose new path selection strategies to improve latency of Tor based on actively measuring latencies and passively observing bandwidths of direct links to Tor relays. Similarly, Snader and Borisov [74] propose tunable path selection algorithms, so that users can choose between strong anonymity protection or better performance. Under the assumption that this choice does not make users partitionable, overall anonymity for all users increases because of an increase in the total number of users.

3.2 Directory System

Clients need a list of all relays including their addresses, public keys, and policies to exit to certain targets in order to build circuits. Therefore, a small number of trusted directory authorities keeps a list of active relays and serves them to clients. It is vital that all clients know roughly the same set of relays. Otherwise, an adversary could easily identify those clients using a different set of relays. Tor contains a separate directory protocol [63] that specifies how routing information is distributed to clients.

The first directory protocol version was designed for single directory authorities that serve directories of all relays to the clients. The major problem of this version is that directories containing all information about a relay, including public keys, grow pretty fast. The first protocol version was therefore extended by *directory caches* which help distributing the load to multiple nodes. Further, *network status documents* were introduced which contain a short list of routers rather than the complete descriptors. This separation allows clients and caches to download network status documents in shorter intervals and request only the missing router descriptors afterwards.

Another problem of the first directory protocol was that clients had to believe in the network status document of a single authority. Therefore, in the second directory protocol version clients downloaded network status documents from all directory authorities and combined them to obtain a common view on the network. However, as the network grew, the network status documents did so, too. This has made it quite expensive for clients to download all network status documents and combine them locally.

At the time of writing, the third directory protocol version [63] is in use. The idea of this version is to combine network status documents already on the directory authorities and serve documents which are signed multiple times to clients. Whenever a relay is (re-)started or has changed its

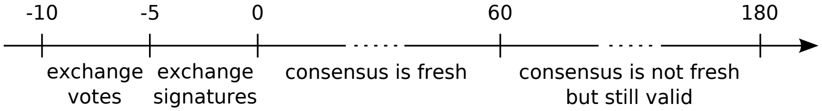


Figure 3.2: Creation and validity of a network status consensus (min)

configuration, it creates a *router descriptor* containing, among other things, its IP address and *onion port* for incoming connections, long-term *identity key*, medium-term *onion key*, and its *exit policy*. The relay then uploads its router descriptor to all directory authorities.[12] The idea is that clients should only use a relay that is known to at least half of all directory authorities; otherwise a single directory authority could fool clients by serving a specifically prepared list of relays in the attempt to destroy their anonymity. As a solution to counter this problem, the directory authorities vote in fixed intervals on a common list of routers, the *network status consensus*. Clients download the consensus from any of the directory authorities to obtain a common view on the network. Afterwards, they need to download the corresponding router descriptors to be able to create circuits.

Figure 3.2 shows the timing of creation and validity of a network status consensus. The times are system-wide defaults at the time of writing and may change in the future. 10 minutes prior to publication of a consensus, the directory authorities start exchanging signed *network status vote* documents containing the identities of known relays. Every directory authority then computes a *network status consensus* document containing the identities of all relays that are listed in the majority of all votes. Under the assumption of full connectivity between directory authorities, every authority comes up with the same list of relays. 5 minutes prior to publication, the authorities exchange detached signatures of the previously created consensus. At time 0 the consensus is published and made available

[12] At the time of writing there are six directory authorities running the described version 3 of the directory protocol.

to clients. It is then *fresh* for 60 minutes at which point a new consensus is published. A consensus is *valid* for 180 minutes after publication during which clients do not need to download a new consensus. After 180 minutes at the latest a consensus is discarded and clients download a new consensus.

3.3 Hidden Services

Hidden services make use of initiator-anonymous circuits to provide responder pseudonymity. The hidden service protocol defines three additional roles that are implemented by Tor relays: *hidden service directory server* (which can be distinct from the directory authorities as described above), *introduction point*, and *rendezvous point*. Tor clients can both provide and access a hidden service. The in-depth specification of the hidden service protocol is described in the Tor rendezvous specification [65], which already includes the changes that have been performed for the purpose of this thesis. Figure 3.3 shows the steps to set up a hidden service in the network and to establish a connection to it.

Before being able to offer a hidden service, service provider Bob generates a public key pair as long-term identity for his service. Bob then establishes his service in the Tor network by randomly selecting a small number of relays as his *introduction points* and creating circuits to them. He sends ESTABLISHINTRODUCE cells containing the public key of the service to the prospective introduction points in step 1. The introduction points acknowledge the request by sending INTROESTABLISHED cells in step 2, meaning that the relays are ready to accept introduction requests from clients.

After establishing a sufficient number of introduction points, Bob creates a *hidden service descriptor* containing the public key of the service, a timestamp, and the introduction point list, signed with the private key of the service. He uploads this descriptor to the three authoritative hidden

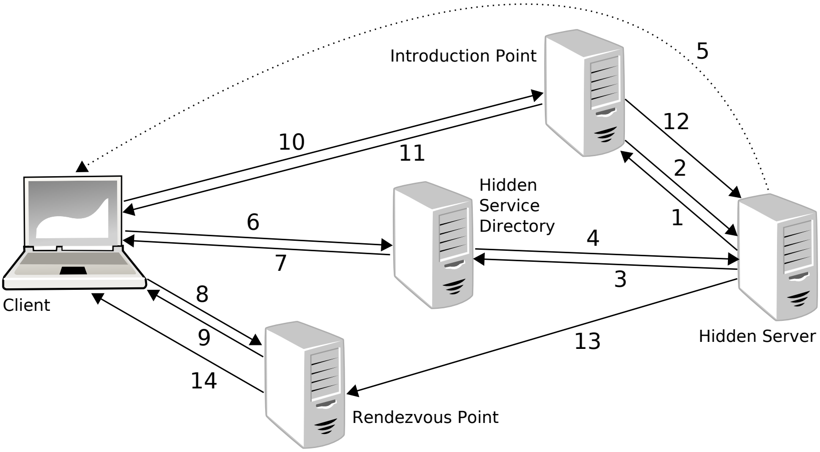


Figure 3.3: Overview of the hidden service protocol

service directories in step 3 which is acknowledged in step 4. Bob uses another circuit for uploading his descriptor in order to hide his IP address from the directories as well. After that, the hidden service is established and ready to be contacted by clients.

Bob can tell the *onion address* of his service, which is a hash of the public key of his service, to his clients in step 5. The advantage of using a service name that is derived from a public key as opposed to a freely selectable name is that it is self-authenticating: clients can verify that an obtained hidden service descriptor was created by the service that they expect and do not have to trust the directory servers in returning the correct descriptor for a given service name. The disadvantage, however, is that onion addresses are less convenient for users to handle than freely selectable names.

A client Alice who wants to establish a connection to Bob's service starts by fetching his hidden service descriptor from the directory servers in step 6 and receives a response in step 7. Alice uses a circuit for this request to hide her attempt to access Bob's service from the directory servers.

If Bob's service is available and Alice has received a hidden service descriptor, she establishes a separate rendezvous point that will be used to transfer user data for her request. Typically, she does not need to establish a new circuit from scratch, but can reuse a preemptively established circuit for this purpose; this process is called *cannibalization* and is always used when a circuit needs to be created on demand. Alice sends an ESTABLISHRENDEZVOUS cell to the prospective rendezvous point including a single-use random string, the *rendezvous cookie*, in step 8. The rendezvous point stores this cookie and acknowledges receipt by responding with a RENDEZVOUSESTABLISHED cell in step 9. In the meantime, Alice establishes a circuit to one of Bob's introduction points. She may reuse an existing circuit by means of cannibalization and extend it by a single hop to the introduction point.

As soon as Alice's rendezvous point and the circuit to Bob's introduction point are established, Alice sends an INTRODUCE1 cell to the introduction point in step 10 containing the unencrypted hash of Bob's public key and an encapsulated message part. The latter is encrypted using Bob's public key and contains IP address and onion port of Alice's rendezvous point, the rendezvous cookie, and the first half of a Diffie-Hellman handshake. The introduction point compares the unencrypted hash with previously received public keys of services. If the introduction point finds a match, it acknowledges Alice's request in step 11 and forwards the encrypted message part of the INTRODUCE1 cell as INTRODUCE2 cell to Bob in step 12. Bob establishes a circuit to Alice's rendezvous point, possibly by extending a cannibalized circuit, and sends to it a RENDEZVOUS1 cell containing the rendezvous cookie and the second half of the Diffie-Hellman handshake in step 13. The rendezvous point, upon recognizing the rendezvous cookie, forwards the second half of the Diffie-Hellman handshake as RENDEZVOUS2 cell to Alice in step 14, finally establishing an end-to-end encrypted circuit between Alice and Bob. Alice can then attach application-level streams to the circuit and perform service requests.

3.4 Threat Model

A threat model defines the capabilities of an adversary which the system can protect its users from. A common threat model in privacy-enhancing technologies is that of a global passive adversary; in this model the adversary is capable of observing (but not modifying) all traffic that is passed within the anonymous communication system including from and to its users. Tor [11], like all other practical low-latency anonymous communication systems, does not protect against a global passive adversary. Tor does not introduce artificial delays in the transported traffic and does not add cover traffic at times when no real traffic is sent. An adversary who is able to observe all traffic between the relays and between clients and relays can link the initiators to responders and hidden services to the servers which provide them. Such an attack would be performed using traffic analysis techniques, that is, by comparing patterns in the observed traffic.

In contrast to this, Tor protects its users against an adversary that can control only a limited fraction of all network traffic. This adversary can generate, modify, delete, or delay traffic, can operate relays of his own, and can compromise some fraction of other relays. The assumption of such an adversary is more realistic in the Tor network. Relays are run by volunteers distributed over the whole world which, however, allows an adversary to become part of the network with one or a few nodes without attracting much attention. In general it is deemed to be sufficient to observe only the ends of a circuit to correlate initiator and responder, or in case of hidden services, to link a hidden service to the server that provides it. Attacks on low-latency anonymous communication systems that make use of traffic analysis are described, for example, by Raymond [66] and Serjantov and Sewell [72].

This chapter has given a brief overview of Tor and its hidden services feature. This description will be necessary to understand the limitations of hidden services for private services and will be required to comprehend the contribution of this thesis. The next three chapters will point out the specific problems of hidden services with respect to implementing private services and will present solutions to overcome them.

4 Distributed Descriptor Storage

Anonymous communication systems that support recipient pseudonyms usually rely on a *directory service*. Its purpose is to store and serve *descriptors* containing the mapping from a long-term pseudonym to short- or medium-term contact information. In most cases the contact information expires after a certain time, but initiators shall still be able to contact the recipient under a persistent pseudonym. Another reason is convenience, so that users do not need to remember cumbersome contact information but can refer to a certain recipient using a human-readable name. The requirements to directories for recipient pseudonymity as described below exceed those to naming services in general when it comes to privacy of stored entries. These requirements make the design of a directory service for this application a non-trivial task.

This chapter presents a novel distributed directory design for Tor hidden services. The basic elements of the design have been described earlier as part of a privacy-aware instant messaging system [39, 40]. The idea was to rely on Tor hidden services to provide the instant messaging service and distribute contact data in a public distributed hash table outside of Tor. The design provided for a way that entries in the distributed hash table are only detectable and comprehensible for intended users. The design presented here adopts many of these ideas, like encoding of descriptor identifiers, and evolves them towards a general distributed hidden service directory.

The chapter is structured as follows: The desirable requirements to naming services are listed in Section 4.1. Previous designs, which are either centralized or decentralized, are described in Section 4.2. Section 4.3

contains a description of the existing Tor hidden service directory design. Section 4.4 contains the proposal of the distributed hidden service directory for Tor. The possible impacts of the new design on security properties are investigated in Section 4.5. An evaluation of the availability of stored descriptors based on a statistical analysis of archived data about the Tor network is presented in Section 4.6. Some facts about the implementation of the distributed storage that has been deployed in the public Tor network are outlined in Section 4.7. Section 4.8 concludes the chapter.

4.1 Requirements

A directory service for long-term recipient pseudonyms has the purpose to store and serve mappings from pseudonyms to contact information. Hence, the functional requirements are the two operations of publishing and retrieving such directory entries. The non-functional requirements can be subdivided into requirements to naming services in general and those requirements which are specific to anonymous communication systems. Clearly, it is hard for any directory solution to fulfill all requirements in equal measure.

Two general requirements are availability and scalability: A major requirement to a directory service is *high availability*, that is, the proportion of time the service is working should be close to 100%. The directory service should not become unavailable when a single or a few nodes in a network fail or are attacked. Further, the directory service should *scale to a large number of requests and entries.* New applications, possibly including private services as described in this thesis, are likely to increase the load of the directory service. High scalability also protects against denial-of-service attacks that attempt to make the system unavailable by overloading it with useless entries.

Specific requirements to directory services for long-term recipient pseudonyms are authenticity of entries, censorship resistance, concealing activity, and support for private entries: The directory should verify *authenticity of stored entries*, so that clients fetching a certain entry can be sure that the entry was created by the service which they specified in their request. Clients should not need to trust the directory in this context, but should be able to verify authenticity themselves. The directory servers should *not be able to censor entries* for a certain pseudonym easily in order to make that recipient unavailable, neither on their own behalf, nor when being forced to. Requests to the directory can reveal a lot of information about a service and its clients: publish requests indicate *service activity*, and fetch requests reveal actual *usage of a service* by clients. While it is very hard to conceal activity completely, request data should not be accumulated at a single place over time. The directory should *support storage of entries for certain clients*, so that only these clients can locate the entry in the first instance, fetch it, and understand the contained contact information.

4.2 Previous Work on Descriptor Storage

Previous work on directory service designs can be subdivided into centralized and decentralized approaches. While the earlier pseudonymous service designs relied on centralized solutions, decentralized designs have been proposed and deployed more recently.

The high-latency anonymous communication systems with support for recipient pseudonymity that rely on a directory service all use a centralized design. These systems include nymservers in the Penet remailer and both Cypherpunk-style [47] and Mixminion-style remailers [8]. In these designs, contact information for recipients is stored on a single server and used to forward incoming messages to the pseudonymous recipients. Albeit being simple, these directory service designs fail to provide all requirements as stated above. Availability depends on a single point of failure,[13] and scalability is limited by the resources that are available to a nymserver. Even though the nymserver might ensure that contact information is authentic, the sender of a message cannot verify this, because the nymserver forwards messages directly, rather than returning entries to the sender. The nymserver might censor any pseudonym by blocking the forwarding of messages and learns about all activities related to a pseudonym.

One of the first directory service designs for low-latency anonymous communication systems, the TAZ Server [23], was a centralized design, too. The difference to a nymserver is that contact information, a so-called *rewebber locator* in this case, is returned to the client rather than forwarding its request. TAZ servers are exposed to the same problems concerning availability and scalability as nymservers. Authenticity of entries is explicitly excluded from the design, so that TAZ servers merely need to provide simple lookup functionality. From this follows that the other requirements, such as censorship resistance, concealing activity, and support for private entries are not included in the design, either.

Tor hidden services [11] rely on a centralized approach, too, which consists of three distinct directory servers.[14] Hidden service descriptors are replicated to all three directory servers which increases availability. Scalability, however, is an issue, because every directory server stores every descriptor, which obviously does not scale for a large number of descriptors. Descriptors are referred to by the hash of the public key that is used to identify the service and to sign the descriptor, so that authenticity of entries is verified by directory servers and clients. The current design does not exhibit censorship resistance, as the three directory servers could de-

[13] There might be multiple nymservers being responsible for different pseudonyms, which, however, does not improve availability of a single pseudonym.

[14] The Tor design paper already suggests to utilize a distributed hash table instead of the centralized storage, which, however, was not implemented in favor of the simpler server-based design.

cide to block a specific descriptor to make the hidden service unavailable. Activity is not concealed from the directory server operators who could analyze these data to derive information about service activity or usage. Private entries are not supported.

The Pseudonymous IP Network [19] was the first design to make use of a decentralized storage, in particular an unstructured peer-to-peer network like Gnutella [18]. This approach solves the requirements of availability and scalability quite well and also exhibits improved security properties. The distributed storage is resistant to censorship, and activity is concealed from a single node operator. Authenticity of entries, however, is not achieved by the Gnutella-based design, as an adversary could store an arbitrary number of false entries under a given pseudonym name. The design also makes no provisions for storing private entries.

The I2P system[15] uses a distributed hash table to store information to contact I2P nodes. This approach ensures availability and scalability. Entries are authenticated by the storing nodes which verify their signatures. Censorship resistance and concealing of activity are ensured by periodically changing responsible storage nodes. Private entries are not specified.

4.3 Existing Tor Hidden Service Directory Design

The previous discussion of designs for directory services has shown that none of them provides all of the requirements as stated above. The centralized approaches lack availability and scalability and do not prevent the operators from censoring or tracking activity of pseudonyms. The decentralized approaches can improve these properties, but none of them fulfills all stated requirements. The goal of this chapter is to present and evaluate a new decentralized directory service for Tor hidden services. Therefore,

[15] See the I2P homepage: `http://www.i2p2.de/techintro.html` (last checked: Dec 17, 2008)

the existing directory service design for hidden services in Tor is discussed first with respect to fulfilling the stated requirements.

The existing Tor hidden service directory design relies on three fixed directory servers. Their addresses and identity keys are hard-coded in the Tor source code and are supposed to change seldom.[16] Hence, the directory servers can be considered to be well-known to all hidden servers and clients.

Hidden servers publish the descriptors for their hidden services to the directory ports of all three directory servers using an HTTP post request. Requests are tunneled via Tor circuits to conceal the hidden server as origin of the request; however, the descriptor content is transmitted in the clear between the last node in the circuit and the directory server.[17] Hidden servers repeat this step whenever their descriptor changes or when an hour has elapsed. Clients that want to access a hidden service try to fetch its descriptor from one randomly selected directory server. This request is also tunneled via a Tor circuit to protect the location of the requesting client. Failed requests are not repeated at other directory servers, unless the user makes another attempt to reach a hidden service.

A hidden service descriptor in the existing design consists of the public key of the hidden service, a timestamp, a list of introduction points, and a signature created with the corresponding private key. Hidden service descriptors are identified by the hash of the public key which is equal to the onion address of the hidden service. Both the directory servers and clients can verify that a hidden service descriptor must have been created

[16] The IP addresses of the three directory servers have changed five times between their initial setup on April 29, 2004 and the time of writing this on September 27, 2008; changes took place on May 7, 2004, October 14, 2004, July 15, 2005, November 5, 2005, and May 23, 2007.

[17] The existing design has been improved in the course of this thesis by extending circuits to the directory servers and sending encrypted directory requests in so-called BeginDir cells.

Table 4.1: Hidden service descriptor format, version 0

Field	Description
Service key	Public service key
Timestamp	Time when descriptor was created
Introduction points	List of introduction points
Signature	Signature of above fields created with private service key

by the owner of the private key which is the hidden server. Table 4.1 shows the format of a version 0 hidden service descriptor as described in the Tor rendezvous specification [65].

Availability of the existing hidden service directory design can be evaluated empirically by considering availability of the three hidden service directories.[18] An evaluation of 1,968 hourly network status consensuses between January 10, 2008 and March 31, 2008 has shown that in 104 cases one of the three hidden service directories has either failed or been restarted. Given that servers usually upload a new descriptor every hour and that clients do not retry failed downloads at other directories, clients were unable to download descriptors for a mean downtime of $30 \times 1/3 = 10$ minutes for every such incident. As a result, the 104 downtimes accounted for 17.33 hours in an interval of 1,968 hours being a ratio of 0.9%. The other way round, the overall uptime of the hidden service directory in the observed time was 99.1%.

Scalability of the design can best be evaluated by considering the average numbers of requests to a hidden service directory. Table 4.2 contains the number of publish and fetch requests per hour between May 1, 2007, 23:25 and May 2, 22:25 UTC (Coordinated Universal Time) to the hidden service directory server `moria1` run by Roger Dingledine. Figure 4.1 vi-

[18] All evaluations based on historical Tor network data have been performed using the Tor network archives collected on the directory server `tor26` by Peter Palfrader. The subsequent statistical analysis was conducted using GNU R [79].

Table 4.2: Hidden service requests per hour to Tor directory server `moria1` between May 1, 2007, 23:25 UTC and May 2, 2007, 22:25 UTC

Request Type	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Publish	1169	1374	1456	1453	1548	1712
Fetch	50	90	103	113	142	207

sualizes these data in a graph. The low number of fetch requests results from the fact that only one third of the fetch requests can be observed at a single directory server; nevertheless, the ratio of requested to provided hidden services is rather low. These data have been collected in an anonymity-preserving way by accumulating the number of requests in time slices of 15 minutes and writing a history of 24 hours to a file on the directory server. There is no public archive of requests available, so that this evaluation should not be considered representative.[19]

The remaining four requirements have already been discussed in the previous section. While the design ensures authenticity of descriptors, the other requirements of censorship resistance, protection of service activity, and support for private entries are not provided.

4.4 Proposed Tor Hidden Service Directory Design

The design of directory servers and hidden service descriptors in Tor meets only a few of the previously stated requirements. In contrast to this, a decentralized approach has a number of advantages which could be leveraged in Tor. Storing hidden service descriptors on a subset of a large number of directory nodes instead of only three servers exhibits better scalability and can also increase availability. Furthermore, it becomes

[19] The data have been posted to the Tor developer mailing list on May 2, 2007: `http://archives.seul.org/or/dev/May-2007/msg00004.html` (last checked: Dec 17, 2008)

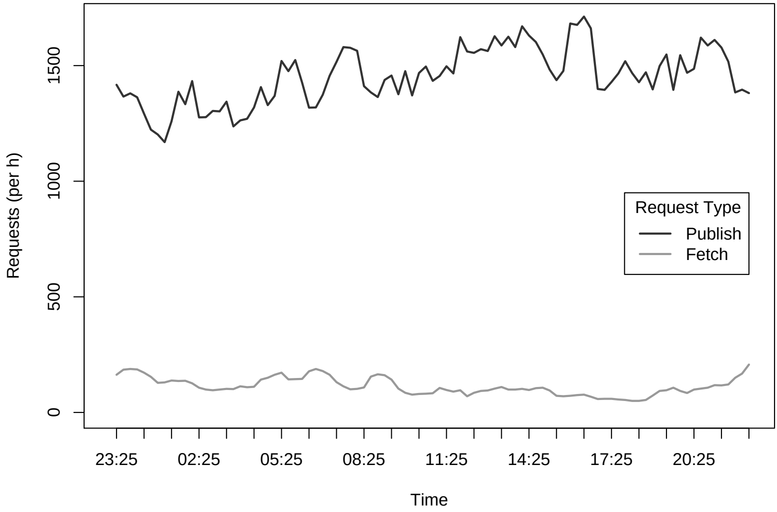


Figure 4.1: Hidden service requests per hour to Tor directory server `moria1` between May 1, 2007, 23:25 UTC and May 2, 2007, 22:25 UTC

significantly harder for a single node to either censor a service or track its activities; even if the operators of the three directory servers are perfectly trustworthy people, they could be forced or subpoenaed to censor a service or reveal information about descriptor publications or fetches. At last, a decentralized approach allows for private entries with even better protection of contact information and activity from unauthorized clients.

Besides the possible benefits of a distributed directory, there are a few threats to keep in mind. Migration of the directory service from the presumably trusted directory authorities to a large number of possibly untrusted nodes might introduce new problems. It must be assumed that an adversary can control a limited part of nodes and connections in the network, thus being able to set up own directory nodes and read or modify messages to other directory nodes. On the one hand, descriptors shall still be signed and the lookup process shall be based on the hash of the public key, which prevents directory nodes from forging descriptors. On the other hand, an adversary might learn information that was not public in the existing design which must be counteracted or at least limited.

4.4.1 Overview

The presented decentralized directory design makes use of peer-to-peer principles. Previous experience with both unstructured [44] and structured peer-to-peer systems [30] was of great help when deciding which system to adapt. The proposed design bears close resemblance to distributed hash tables [1], especially Chord [75, 76]. Storage nodes and descriptors in a distributed hash table are assigned unique identifiers determining on which node a descriptor is stored. The nodes maintain routing tables containing a small portion of other nodes and can forward requests to the responsible node.

However, in contrast to general-purpose distributed hash tables, the proposed design makes arrangements to limit the impact of malicious directory nodes: neither routing information nor stored entries are exchanged between directory nodes. While both functions are required for a general-purpose distributed hash table to operate, they would open up security problems that are incompatible with the stated requirements.

Instead of maintaining their own partial routing tables, all hidden servers and clients rely on the common network status consensus. The network status consensus contains a list of all relays in the network including potential directory nodes. It is available on all Tor nodes anyway for performing the actual onion routing. The consensus is updated once an hour by the directory authorities and remains valid for three hours.[20] Directory nodes do not need to exchange routing information or forward directory requests to other nodes. Therefore, they do not need to believe in what

[20] These values reflect the situation at the time of writing this thesis; both voting interval and number of periods for which a network status consensus is valid might change in the future.

other directory nodes say. In return, the directory nodes need to be very stable, so that routing information at hidden servers and clients is as consistent as possible.

Directory nodes do not replicate stored entries among themselves, but rely on the hidden servers to make sure that descriptors are replicated to a sufficiently large number of directory nodes or notice that responsibility for a given descriptor has changed. Consequently, entries are not transfered between directory nodes in case of joining or leaving nodes. This design prevents routing attacks as described in [73] where an adversary joins the system under changing identities and requests replicas.

The subsequent description is split into three parts which are shown in Figure 4.2: A list of current directory nodes is compiled on the directory authorities and distributed to hidden servers and clients in steps 1a and 1b. Hidden servers publish the hidden service descriptors for their services to the directory nodes which verify and store them locally in step 2. Clients fetch hidden service descriptors from the directory nodes in step 3.

4.4.2 Distribution of Consistent Routing Information

Clients and hidden servers require consistent routing information, so that clients fetch descriptors from the directory nodes to which they were published before. The two selection criteria for directory nodes in the proposed design are to serve as Tor relays and to be connected to the Tor network for a *minimum uptime* of a given number of hours. These criteria narrow down the selection of nodes to highly available nodes only, thus ensuring a certain level of consistency of routing information at hidden servers and clients. There is no official authorization necessary to run a directory node. Relay operators can decide for themselves whether to participate in distributed descriptor storage or not by configuring their relay accordingly. The idea is to have hundreds of directory nodes with each of

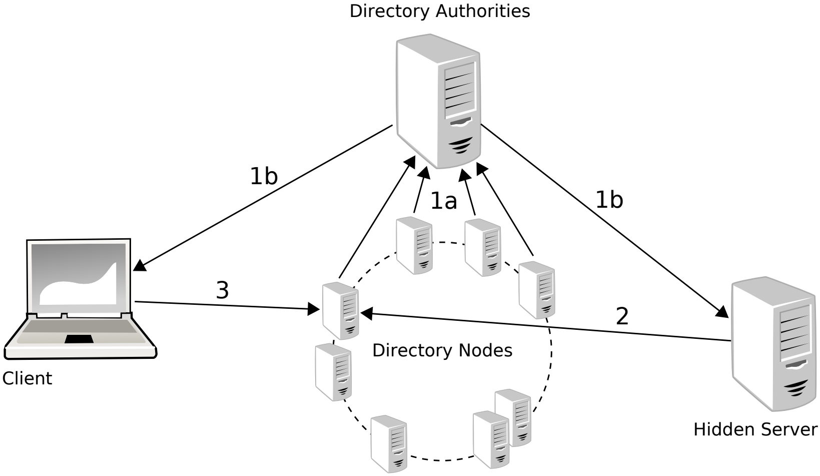


Figure 4.2: Overview of distributed directory in the proposed directory design

them storing and serving a small share of all descriptors. The existence of bad directory nodes shall be countered by means of replication.

Compiling and distributing a list of directory nodes is performed in multiple steps: In the first step, relays that are configured as hidden service directory nodes include a new flag in the router descriptors that they upload to the directory authorities stating their willingness to act as directory node. Next, the directory authorities ensure that they can connect to the relay for the globally defined minimum uptime before assigning the new flag to the relay in their votes and in the consensus. Finally, clients and hidden servers download the network status consensus and can filter all relays with the new flag to obtain the complete list of directory nodes.

Table 4.3: Proposed hidden service descriptor format, version 2

Field	Description
Descriptor identifier	Hash of service identity and secret identifier part
Service key	Public service key
Secret identifier part	Hash of time period, descriptor cookie, and replica index
Timestamp	Time when descriptor was created
Introduction points	List of introduction points (possibly encrypted)
Signature	Signature of above fields created with private service key

4.4.3 Publication of Hidden Service Descriptors

Hidden servers start advertising their hidden services in the network by calculating current descriptor identifiers.[21] As opposed to the existing design, descriptor identifiers change periodically, so that descriptors are stored on changing directory nodes over time. The main reason for this is that directory nodes shall only be responsible for a given descriptor for a limited time. Table 4.3 shows the proposed hidden service descriptor format. A descriptor identifier is calculated by applying a secure hash function H to the concatenation (denoted as ||) of the public key identifier of a hidden service (its onion address) and a possibly secret identifier part. Throughout this thesis SHA-1 [55] is used as secure hash function. The secret identifier part is the result of a hash operation, too.

```
descriptor-id = H(public-key-id || secret-id-part)
secret-id-part = H(descriptor-cookie || time-period || replica-index)
```

The *descriptor cookie* is an optional secret key that is shared between a hidden server and a client. By including the descriptor cookie in the

[21] Assembly of descriptor identifiers in the presented design has certain similarities to an approach by Øverlier and Syverson [57]. Their work was simultaneously published with a previous design [39] that is the basis for the design presented in this thesis. The approach by Øverlier and Syverson will be described in more detail as related work in Chapter 7.

descriptor identifier, the identifier is made unguessable for unauthorized clients. The result is support for private entries.[22]

The *time period* contains the number of the period since the epoch (January 1, 1970, 00:00:00 UTC) for a globally fixed period length of, for example, 24 hours. It is constructed in a way that transition times of descriptor identifiers are equally distributed over the whole period depending on the public key of the service. Preventing descriptor identifiers from changing at the same periodic time avoids re-publication bursts of all descriptors in the system. During period change-over times, hidden servers are required to publish descriptors for the expiring as well as the upcoming period in order to compensate clock deviations.

The *replica index* is used to assign distinct identifiers to the replicas of a descriptor to distribute them to different parts of the identifier range. As opposed to the distributed hash table Chord [76], replicas are not stored on the siblings of a responsible node, but under distinct identifiers. Even though storage on adjacent nodes in combination with node-to-node replication would have advantages in compensation of node failures, it opens the possibility for a very effective threat: an attacker could create a number of directory nodes with subsequent identifiers, for example by performing a Sybil attack [13], large enough to literally create a "black hole" in the identifier circle. Hidden services with descriptor identifiers in the affected identifier range would become inaccessible. Furthermore, directory nodes shall not be responsible for replication anyway in order to restrict dissemination of descriptors. Therefore it does not make a difference whether replicas are stored on adjacent nodes or distributed over the identifier circle.

After determining the descriptor identifiers, a hidden server determines which directory nodes are responsible for storing the descriptor replicas.

[22] This feature is only mentioned at this point for the sake of completeness. It will be discussed in more detail in the next chapter on client authorization for pseudonymous services.

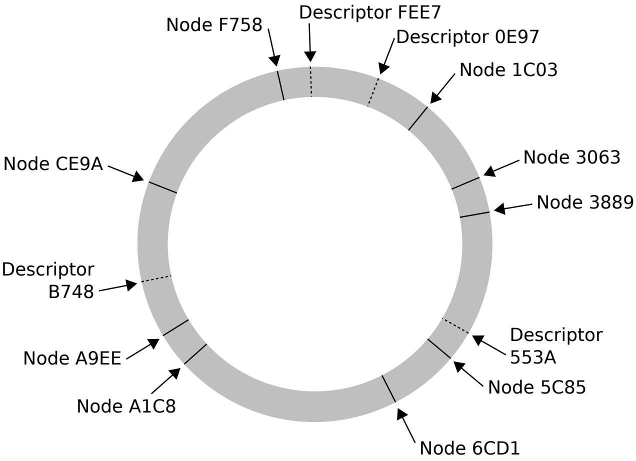


Figure 4.3: Example of distributed storage ring in the proposed directory design

Responsibility of a directory node for certain descriptors is defined by its *node identifier*. In the proposed design, the unique identity keys of relays have been chosen as node identifiers, because they usually do not change over time. Directory nodes are arranged using their identifiers in a closed identifier ring, comparable to a Chord ring [76]. Descriptors have identifiers in the same identifier space as directory nodes. A descriptor is stored depending on its identifier on the directory node with next greater or equal identifier. Figure 4.3 shows an example directory consisting of nine directory nodes and four stored descriptors. In this example descriptor 553A would be stored on node 5C85, descriptor B748 on node CE9A, and descriptors FEE7 and 0E97 on node 1C03.

After having determined the responsible directory nodes for the descriptors of a hidden service, the hidden server creates or extends circuits to the directory nodes. Circuit extension by one or more hops is more expensive than sending an HTTP request from the exit node of a circuit to a directory node. But it hides the descriptor content from the exit node and conceals the fact that a descriptor is published from the exit node. Hidden servers re-publish descriptors whenever their content changes or when they detect that a new directory node has become responsible for storing a descriptor. Failed requests are by no means forwarded to other directory nodes, even if a requested directory node knows that another node might be better suited to store the descriptor, in order to limit unrestrained dissemination of descriptors.

The directory nodes, upon receiving a publish request, need to verify that a hidden server is allowed to store a descriptor under a given identifier. If descriptor identifiers are computed as described above, they look like random strings to a storing directory node. However, without being able to verify the legitimacy of a storing hidden server, anyone could claim to store a descriptor under a given identifier, possibly occupying the slot of a legitimate hidden service. Unfortunately, a hidden server cannot easily move on to a new slot, besides changing the public key and thereby the onion address of a hidden service. Another, superficially neat idea is to store all descriptors for a given identifier without verifying them, because a client would have to download them all to filter out the legitimate ones. This is a race that clients would inevitably lose.

Therefore, hidden service descriptors contain the *secret identifier part* that was used to create the descriptor identifier. A directory node can verify legitimacy for using a descriptor identifier in two steps: First, the directory node verifies the signature of the descriptor content with the included public key. Second, the directory node generates a descriptor identifier using public key and time period and compares it with the claimed descriptor identifier. If both identifiers match, the directory node memorizes the descriptor and makes it persistent. Although a relay would not be selected as directory node for one publication period after restarting, clients might still request descriptors from them for a certain time after restarting.

In contrast to the current design, the novel design makes barely trusted relays responsible for storing hidden service descriptors. While they cannot alter the signed contents, they could deny existence of previously stored descriptors in the attempt of making a hidden service unavailable. As a countermeasure hidden servers periodically try to download their own descriptors. If they fail, they file a complaint against the directory node to the directory authorities. The authorities first re-publish the affected descriptor and then try to fetch it in random intervals. If they can confirm the misbehavior, they penalize the directory node first by temporarily removing its directory node flag. In case of recurrence the directory authorities ban the IP address range of the relay for acting as directory node in the future.

4.4.4 Fetching Hidden Service Descriptors

When fetching descriptors, clients determine the descriptor identifiers for a given onion address and look up the responsible directory nodes in the same way as hidden servers do. The requesting client creates or extends a circuit to one of the responsible directory nodes and sends a fetch request to it. If the directory node has a descriptor for the requested identifier, it returns this descriptor to the client which verifies and uses it to establish a connection to the hidden server. If a request fails for any reason, clients retry at the other responsible directories until either one of their requests succeeds or none of the responsible directory nodes are left to try. Failed requests are not forwarded to other directory nodes in order to limit dissemination of the information that a certain descriptor is requested.

4.5 Security Implications

Distributing the hidden service directory from a fixed set of servers to a subset of a large number of nodes has security implications that need to be discussed. The following list of possible security problems is based on the previously stated requirements to a directory service and a list of possible attacks on distributed hash tables by Sit and Morris [73].

Forge Descriptors. An adversary could want to replace a valid descriptor with an own descriptor in order to impersonate a hidden server. This attack is neither possible in the existing, nor in the proposed design, because descriptors are signed with the private key of a hidden service.

Disseminate False Routing Information. An adversary could try to distribute false routing information to hidden servers or clients in the attempt to target their ability to address publish or fetch requests to the responsible directory nodes and make services unavailable. As discussed in [73], an adversary in a distributed hash table could forward lookups to incorrect or non-existing nodes, send incorrect routing table updates to other nodes, or attempt to partition the network by providing false routing information to bootstrapping nodes. Although these attacks constitute a threat in distributed hash tables, they are not possible in the proposed design where all participants rely only on centrally provided routing information. For a successful routing attack, the majority of directory authorities that create the network status consensuses would have to collude or be compromised.

Cause Frequent Routing Information Changes. In a distributed hash table, nodes can join and leave the system at any time, resulting in maintenance messages to update routing tables. A malicious node could exploit this rebalancing, resulting in a lot of unnecessary traffic and possibly destabilization of the directory. The proposed design is immune to this attack, because routing tables are only updated once an hour based on the network status consensus.

Harvest Service Identities and Access Services. A malicious directory node could accumulate knowledge about (previously unknown) hidden services over time and try to access them. This security risk is unavoidable with the directory nodes being able to understand the contents of descriptors. At first sight the existing design prevents this attack, because service identities are only made known to the allegedly trusted directory servers. However, an adversary could also run a relay and wait to be picked as introduction point or as exit node in a circuit that is used to send a request to a directory server to learn about service identities. After all, hiding the existence of a hidden service was not stated as a requirement in the original Tor design [11].

Track Service Activity and Usage. An adversary controlling a directory node could want to track activity and usage of a specific hidden service. The adversary would conclude service activity from the existence of a locally stored descriptor and usage from requests for that descriptor. Therefore, the adversary would determine future descriptor locations at least one publication period in advance and start a node with an identity so that it would consequently become responsible for at least one descriptor replica. The fact that a directory node is only responsible for a small share of all descriptors impedes analysis of publish and fetch requests over a longer period to derive information on service activity and usage, respectively. Furthermore, the required minimum directory node uptime prevents malicious directory nodes from quickly changing their identity and as a result the identifier range which they are responsible for. Minimum uptime is enforced by the directory authorities which require a directory node to be available for at least one publication period before listing it as directory node.

A similar attack on tracking service activity is also possible in both the existing and in the proposed design by repeatedly requesting a descriptor from the directory servers or responsible directory nodes. In both designs an adversary could also run a relay in the hope of being picked as introduction point or as exit node in a circuit that is used to send a request to a directory server and learn about service activity and usage.

Censor Specific Service. A group of colluding directory nodes can calculate future descriptor identifiers of a hidden service and generate identity keys to become responsible for all respective parts of the identifier ring one publication period in advance. They can then make the hidden service unavailable by not serving its descriptors anymore. This attack requires to run a number of directory nodes that is twice the number of stored replicas: While half of the directory nodes block current descriptors, the other half awaits becoming responsible for the descriptors in the upcoming time period. Clearly, this is a very serious attack, but the ability to report bad directory nodes constitutes an effective countermeasure to it.

Swamp Directory Node with Incorrect or Useless Descriptors. An adversary could swamp a directory node with correct, but useless descriptors for non-existing hidden services. Although such an attack is also possible in the current design, single directory nodes might be more vulnerable to it as opposed to the central directory servers. However, the impact of this attack on overall descriptors availability is limited in a distributed storage system where more nodes need to be attacked as compared to a centralized system. A possible countermeasure could be to make descriptor publications computationally expensive to make the attack unattractive.

Abuse Storage Space. An adversary could attempt to abuse storage space of hidden service descriptors for unrelated data by encoding these data in the contact information part of descriptors. This attack cannot be prevented completely even when checking plausibility for all descriptor parts. But the attack can be made unattractive by limiting descriptor size to a reasonable value. The maximum allowed size for descriptors in the proposed design is 20 kilobytes.

4.6 Evaluation

Most properties of the proposed design can be evaluated analytically, like censorship resistance and tracking of activity. Evaluating the size of the directory as well as availability of stored descriptors, however, is different, because both rely on network characteristics like the number of relays in the network and change rates of the node population. There are several reasons why correctly stored descriptors might be unavailable for clients: Directory nodes could fail or leave the network at any time. Joining directory nodes might only be known to some but not all participants. Malicious nodes might accept descriptors but not hand them out to requesting clients. Therefore, the novel design must resist a certain amount of failure, which is accomplished by means of replication.

In the following, the proposed distributed directory design is evaluated based on empirical data. At first, two general network metrics are measured, namely theoretical *directory size* in terms of participating directory nodes and *churn rates* as the fractions of joining and leaving nodes compared to the whole node population. In a second step, theoretical *descriptor availability* is analyzed which is the probability that a random descriptor can be successfully downloaded by clients. The goal of this evaluation is to show the feasibility of the proposed design as well as to determine useful parameters for the *minimum uptime* that is required for relays to become directory nodes and *number of replicas* that need to be stored for each descriptor.

4.6.1 Network Characteristics

The first part of this evaluation covers general characteristics of the Tor network which have an influence on the distributed directory: directory size and churn rates. The directory size as the number of nodes that participate in the distributed directory affects scalability, censorship resistance, and concealing activity. The more nodes participate in the directory, the better. However, the minimum uptime requirement shrinks the distributed directory by excluding less available nodes. Another assumption is that the minimum uptime affects churn rates: the higher the minimum uptime requirement is, the lower are the churn rates; the rationale behind this assumption is that nodes which are available for some time are more likely to stay connected in the future. These hypotheses are to be evaluated.

Both directory size and churn rates can be evaluated by analyzing archived network status consensuses and router descriptors. For the given evaluation, the network status consensuses and router descriptors in the interval from January 10, 00:00 to March 31, 2008, 23:00 UTC have been analyzed. During this period of 82 days, the directory authorities published 1,968 consensuses containing the total number of 15,697 distinct server identities. Only running nodes were considered, that is, those relays having the `Running` flag set. The first 50 and last 3 consensuses were omitted, so that all investigated consensuses can meet minimum uptime requirements of up to 48 hours. After that, a total number of 1,915 considered consensuses remains. From the 331,862 referenced descriptors there are 11 missing in the archives. A manual check has confirmed that the corresponding relays have been restarted shortly before the consensus was created in which they were missing.

Table 4.4 shows directory sizes as a function of different minimum uptimes. When no minimum uptime is required, there are 1,423 nodes in the network in the mean. For a minimum uptime of 4 hours, thus excluding all nodes that have been running for 3:59 hours or less, the number of relays shrinks by 247 nodes to 1,149 nodes in the mean. It is no surprise that the number of relays shrinks even further when requiring higher minimum uptimes. However, the differences in network sizes

Table 4.4: Number of relays as a function of minimum uptime (h)

Minimum Uptime	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0	1258	1358	1423	1423	1484	1660
4	1019	1113	1150	1149	1185	1274
8	855	963	998	994	1026	1095
12	753	855	885	882	911	971
16	683	773	796	795	818	874
20	626	705	728	725	748	795
24	584	656	683	677	698	739
30	543	616	643	637	659	704
36	510	579	610	603	626	676
48	451	524	556	548	573	618

also shrink with increasing minimum uptimes, so that the difference of network sizes between 20 and 24 hours is only 48 nodes in the mean. After a required minimum uptime between 20 and 24 hours, the mean network size is halved as compared to no minimum uptime requirement.

Figure 4.4 plots the populations of directory nodes for different minimum uptimes over time. The five topmost graphs exhibit daily fluctuations which cannot be observed as clearly for the five bottommost graphs. A possible explanation for this is that a certain number of relays is run on computers which are shutdown over night or connected via dial-in lines that are automatically disconnected daily by their ISPs. The maximum values in the number of relays occur around 17:00 UTC while the minimum values are reached at around 04:00 UTC. While the exact reasons for this phenomenon are subject to further studies, the insight for this evaluation is that node fluctuation suddenly decreases for minimum uptimes of 20 hours or more. These results indicate that a minimum uptime of 20 hours or higher has good characteristics for the proposed distributed directory.

Comparing churn rates is even more useful in the attempt to evaluate

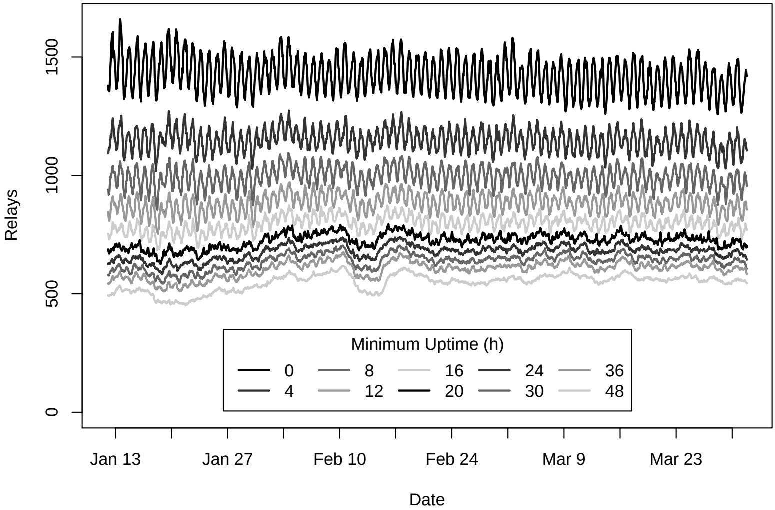


Figure 4.4: Total number of relays for different minimum relay uptimes

short-term fluctuations of node populations. The *join rate* is defined as the fraction of newly joined nodes in a snapshot compared to all nodes in that snapshot. Likewise, the *leave rate* is defined as the fraction of leaving nodes in a snapshot compared to the previous snapshot in which they have been present. The two different reference snapshots (considered or previous snapshot) result from normalization of the metrics to the range of 0 to 100%.

Tables 4.5 and 4.6 show join and leave rates, respectively. Both join and leave rates continually decline with increasing minimum uptime requirements. When no minimum uptime is required, around 4.7% of the nodes leave the network and an equal number of other nodes join within one period of time (here: one hour). Both join and leave rates improve towards a more stable node population for increasing minimum uptimes. With a required minimum uptime of 24 hours, both rates are around 1.1%. After that, both rates decrease only gradually for minimum uptimes of 30 and

Table 4.5: Join rate (%) as function of minimum uptime (h)

Minimum Uptime	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0	1.64	3.61	4.52	4.70	5.71	10.64
4	1.18	2.88	3.71	3.77	4.55	8.06
8	0.72	2.33	3.02	3.07	3.74	6.86
12	0.45	1.98	2.59	2.69	3.29	6.35
16	0.20	1.65	2.28	2.36	2.93	6.13
20	0.17	1.23	1.71	1.80	2.29	5.09
24	0.00	0.65	1.01	1.10	1.44	4.29
30	0.00	0.53	0.86	0.97	1.30	3.93
36	0.00	0.46	0.79	0.90	1.23	3.76
48	0.00	0.30	0.62	0.73	1.01	4.55

36 hours. A minimum uptime of 48 hours exhibits the lowest churn rates of around 0.7%.

Figure 4.5 visualizes these results using box-and-whisker plots [79]. The box part contains all values within the second and third quartile of the data set with the strong line being the median. The dashed lines contain non-outlier values in an interval of 1.5 times the inter-quartile range below the first quartile and the same distance above the third quartile. Outliers are depicted as circles.

As an intermediate result, minimum uptimes of 24 hours or more seem to be most promising for realizing the proposed distributed directory because of the low churn rates. However, even though higher minimum uptimes would further reduce churn rates, they did the same to directory size, thus increasing the share of descriptors a single directory node is responsible for. There is a trade-off between lower churn rates (better descriptor availability) and larger directory sizes (better distribution of trust).

Table 4.6: Leave rate (%) as function of minimum uptime (h)

Minimum Uptime	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0	0.79	3.89	4.70	4.69	5.43	10.03
4	1.40	3.13	3.69	3.77	4.32	7.23
8	0.87	2.41	2.95	3.05	3.62	8.18
12	0.60	2.01	2.58	2.67	3.21	7.37
16	0.42	1.69	2.26	2.34	2.84	6.91
20	0.13	1.26	1.71	1.79	2.25	5.29
24	0.00	0.66	0.99	1.09	1.46	3.87
30	0.00	0.52	0.88	0.97	1.32	4.15
36	0.00	0.46	0.80	0.90	1.23	4.82
48	0.00	0.30	0.62	0.73	1.02	3.81

4.6.2 Descriptor Availability

The second part of this evaluation deals with *descriptor availability* in the proposed distributed directory. Descriptor availability is defined here as the percentage of descriptors that a client can possibly fetch within a period of one hour. If the set of directory nodes would never change and nodes would never fail or be restarted, all previously stored descriptors could be downloaded by clients and descriptor availability would be 100%. There are at least two reasons for a reduction of descriptor availability: different consensuses and restarted directory nodes.

Client and hidden server might work with *different network status consensuses*, thus having different views of the network. Consensuses are valid for three periods, and neither servers nor clients are required to use a fresh consensus for publishing or fetching hidden service descriptors. Consequently, a hidden server could store a descriptor on a directory node that is unknown to the client, or the client could request a descriptor from a directory node that is unknown to the hidden server. As a result, both leaving and joining nodes reduce descriptor availability.

Another reason for descriptor unavailability are *restarted directory nodes*.

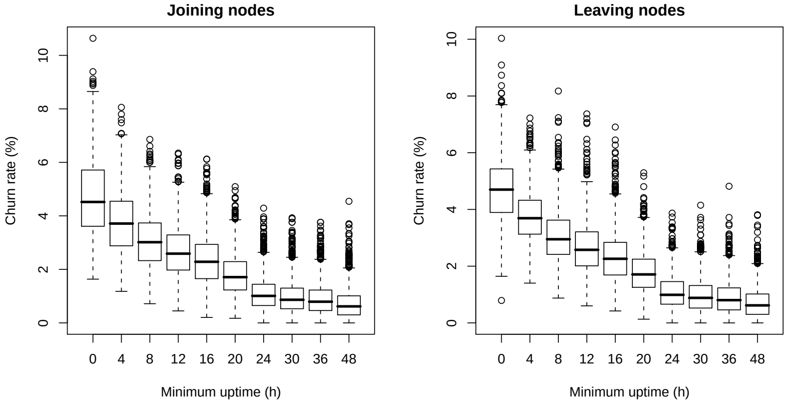


Figure 4.5: Churn rates as functions of minimum uptime

Even though the directory nodes make descriptors persistent, so that they would survive a restart, a restarted node could have been unavailable during either the publish or a fetch request. Restarted nodes show up in the consensuses as nodes that leave the network and rejoin within the same period. While most restarted nodes rejoin immediately, there is no way to determine the exact node downtime. Therefore, in a conservative estimation, restarted directory nodes need to be counted as failing nodes, reducing the descriptor availability, too.

Figure 4.6 contains an example for determining descriptor availability where the hidden server uses an older consensus for storing a descriptor than the client uses to fetch it. In this example, node 6CD1 has left the network and is not contained in the consensus used by the client. While the server would have stored descriptors with identifiers between 5C86 and 6CD1 on node 6CD1, the client would attempt to fetch them from node A1C8. Likewise, node BA3B has joined the network without knowledge of the server, but as part of the consensus that is used by the client. The server would have stored all descriptors in the interval of A9EF to CE9A on node CE9A, but the client would expect descriptors with identi-

Server	Client
1C03	1C03
3063	3063
3889	3889
5C85	5C85
6CD1	A1C8
A1C8	A9EE
A9EE	BA3B
CE9A	CE9A
F758	F758

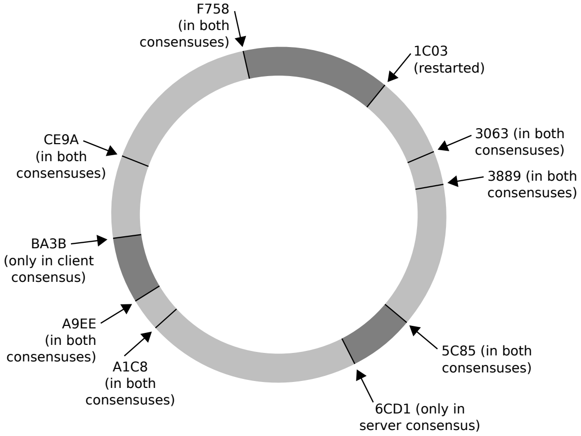

F758 (in both consensuses)
1C03 (restarted)
3063 (in both consensuses)
3889 (in both consensuses)
CE9A (in both consensuses)
BA3B (only in client consensus)
A9EE (in both consensuses)
A1C8 (in both consensuses)
5C85 (in both consensuses)
6CD1 (only in server consensus)

Figure 4.6: Example for determining descriptor availability using two different network status consensuses

fiers between A9EF to BA3B to be stored on node BA3B. In the example, node 1C03 was restarted before the end of the considered period, so that descriptors in the interval from F759 to FFFF and from 0000 to 1C03 are considered as unavailable, too. The descriptor availability, as the share of available identifiers in the whole identifier ring, is 72.9% in this example.

The descriptor availability in the period from p to $p + 1$ depends to a large extent on the compared consensuses. Both client and server could use three different consensuses to determine responsible directory nodes: A fresh one that was recently published at time p, one that was published at time $p - 1$ and is at least one period old, or one that was published at time $p - 2$ and is already more than two periods old.[23] This leads to a

[23] There are at least two cases in which either server or client could use even older consensuses than that of $p - 2$: First, the system clocks of either client or server could be slow, so that an invalid consensus of $p - 3$ might still be in use; this is a rare case and will not be considered to determine useful system parameters here. Second, the server could have published a descriptor based on the consensus of $p - 3$ or earlier, but has confirmed that the changes in $p - 2$ or later did not affect responsibility of the descriptor; in this case it

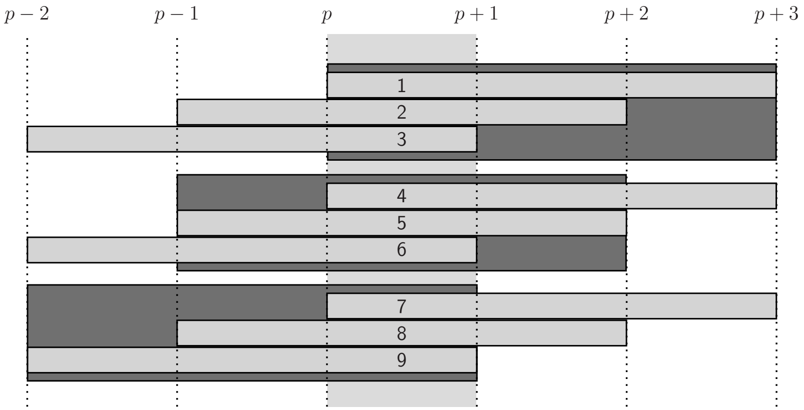


Figure 4.7: Possible combinations of consensuses used by hidden server (dark gray) and client (light gray)

total number of 9 consensus combinations as shown in Figure 4.7. These combinations will be referred to using the numbers 1 (client and server both use consensus p) to 9 (client and server both use consensus $p-2$).

From the 9 possible combinations, three pairs behave similarly when it comes to descriptor availability: these are combinations 2 and 4, 3 and 7, and 6 and 8. When considering two consensuses S published at p_S used by the server and C published at p_C used by the client, this symmetry can be explained: It does not make a difference for calculation of descriptor availability whether p_S precedes p_C and a relay has joined in C, or p_C precedes p_S and the relay has left in S. In both cases descriptors in the identifier range that the relay is responsible for are not available to clients. Further, nodes that have been restarted between p_S and $p+1$ are counted as unavailable, too. Surprisingly, reference to S in this calculation does not hurt symmetry: If a relay has been restarted after p_S but before p_C, it is not included in C, because it does not meet the minimum uptime

is safe to assume that the server could also have used $p-2$, $p-1$, or p for publication.

requirement. The other way round, if a relay has been restarted after p_C but before p_S, it is not included in S for the same reason. If a relay has been restarted after both p_S and p_C, it is safe to consider only S to determine if a relay has been restarted between p_S and $p + 1$. As a result, the number of consensus combinations can be reduced to six.

Table 4.7 shows descriptor availabilities of the consensus combinations with a fixed minimum relay uptime of 24 hours. Figure 4.8 displays these data in a box-and-whisker plot. These results confirm the intuition that the consensus combination has a significant influence on descriptor availability. In particular there are two main observations: The first observation is that the higher the distance of time between server and client consensus is, the lower is the average descriptor availability. For example, the mean descriptor availability of combination 3 is worse than that of combination 2 which in turn is worse than that of combination 1. The same applies to the combinations 4 to 6 and 7 to 9 with combinations 5 and 9 having the best descriptor availabilities, respectively. This decrease in descriptor availability can be explained by the fact that the effects of joining and leaving relays accumulate with higher time distance of consensuses.

The second observation is that the farther a server consensus lies in the past, the worse is its average descriptor availability for a fixed client consensus. For example, the average availabilities of combinations 1, 4, and 7 are in each case decreasing. This effect can be explained by accumulation of restarted nodes between the creation time of the consensus used by the server and $p + 1$.

The consensus combination must be taken into account when performing this evaluation, but cannot be controlled by system parameters (besides changing the number of periods that a consensus is valid, which is not really an option). In the further analysis, a single representative consensus combination shall be fixed in order to keep the evaluation simple. Combination 7 exhibits the worst performance in terms of descriptor availability. It is therefore suitable for finding conservative parameters

Table 4.7: Descriptor availabilities (%) depending on consensus combination (1 to 9) with minimum relay uptime of 24 hours

Consensus combination	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1	95.71	98.54	98.98	98.89	99.34	100.00
2, 4	91.55	96.13	96.90	96.78	97.56	99.37
3, 7	88.39	93.95	94.93	94.82	95.81	98.32
5	93.15	97.33	97.94	97.85	98.51	99.83
6, 8	90.21	95.01	95.92	95.78	96.69	98.91
9	91.33	96.18	96.96	96.84	97.66	99.58

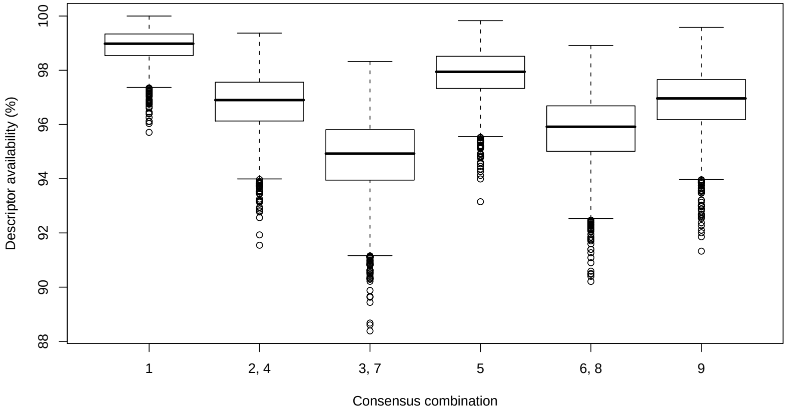


Figure 4.8: Descriptor availability as a function of consensus combination

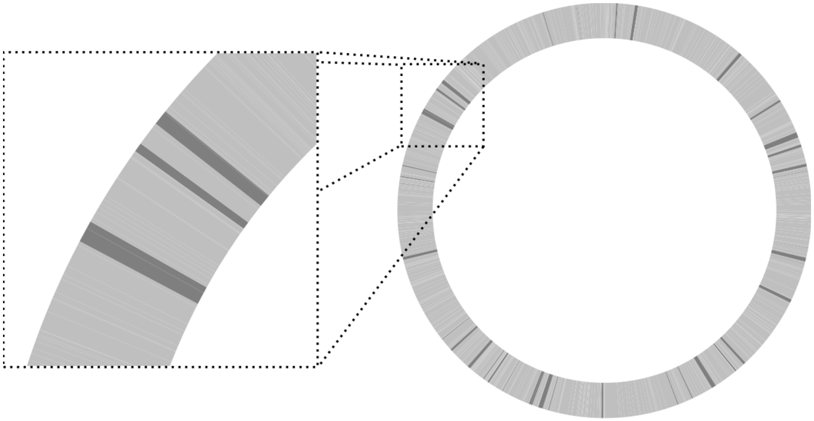


Figure 4.9: Descriptor availability in the period from Feb 14, 2008, 14:00 to 15:00 for consensus combination 7 and minimum uptime of 24 hours

for minimum uptime and number of replicas. From this follows that in the following discussion the server consensus is already two periods old, whereas the client consensus has been freshly published.

Figure 4.9 shows an example of the descriptor availability for combination 7 and a minimum uptime of 24 hours. The figure visualizes descriptor availability of the time between Feb 14, 2008, 14:00 and 15:00 with the server consensus from Feb 14, 12:00 and the client consensus from Feb 14, 14:00. The resulting descriptor availability is 93.8% (parts in light gray). Loss of descriptor availability (parts in dark gray) results from leaving (2.3%), joining (2.8%) and restarted nodes (1.1%). In total there were 645 relays meeting the minimum uptime requirement in the 12:00 consensus and 657 in the 14:00 consensus.

Beyond measuring the share of available identifier ranges, the share of unavailable ranges can be subdivided into the three possible reasons for descriptor unavailability: joining, leaving, and restarted nodes. Table 4.8 splits up reasons for descriptor unavailability for the fixed consensus combination 7 and a minimum relay uptime of 24 hours. While both joining

Table 4.8: Shares of unavailability (%) for different reasons with fixed consensus combination 7 and minimum relay uptime of 24 hours

Reason	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Joining Nodes	0.18	1.48	1.98	2.09	2.62	5.29
Leaving Nodes	0.17	1.46	2.00	2.11	2.60	6.79
Restarted Nodes	0.00	0.55	0.89	0.99	1.33	4.27

and leaving nodes account for about 2.1% of descriptor unavailability each, restarted nodes make only about 1.0% of the descriptors unavailable.

The next step is to evaluate the influence of minimum uptime on descriptor availability. If joining and leaving nodes account for the major part of descriptor unavailability and churn rates shrink with higher minimum uptimes (as shown in the last section), higher minimum uptimes should also result in higher descriptor availability. Table 4.9 shows descriptor availabilities as a function of minimum uptimes for the fixed consensus combination 7. When no minimum uptime is required, descriptor availability is at only 80.6% in the mean, which means that about 1 out of 5 descriptors gets lost. Descriptor availability grows quickly with higher minimum uptime requirements to 94.8% for 24 hours minimum uptime. After that, descriptor availability grows less quickly to 96.4% for a minimum uptime of 48 hours. Figure 4.10 displays the data using box-and-whisker plots. These results confirm the intermediate finding that a minimum uptime of 24 hours is a good trade-off between descriptor availability and directory size.

For a fixed minimum uptime and therefore descriptor availability, it is possible to determine the number of replicas that are required to ensure a certain availability of a descriptor. The more replicas are stored, the higher is the probability that at least one replica of a descriptor remains available and the service can be accessed. But a higher number of repli-

Table 4.9: Descriptor availabilities (%) depending on minimum uptimes (h) for fixed consensus combination 7

Minimum Uptime (h)	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0	70.85	77.84	80.72	80.57	83.33	88.20
4	74.60	81.37	83.22	83.15	85.00	89.48
8	79.30	84.54	85.95	85.90	87.39	92.23
12	79.22	86.18	87.56	87.53	89.02	93.08
16	79.44	87.38	88.99	88.95	90.68	94.83
20	85.23	90.29	91.60	91.50	92.84	96.34
24	88.39	93.95	94.93	94.82	95.81	98.32
30	88.85	94.46	95.34	95.28	96.25	99.09
36	88.96	94.84	95.73	95.62	96.56	99.40
48	88.94	95.62	96.60	96.44	97.43	99.47

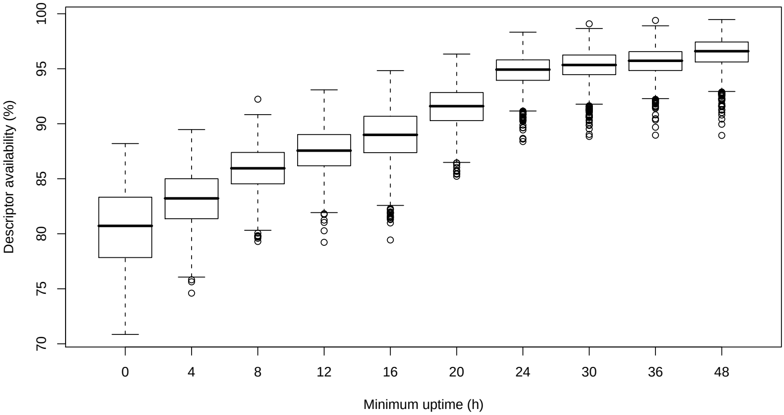


Figure 4.10: Descriptor availability as a function of minimum uptimes

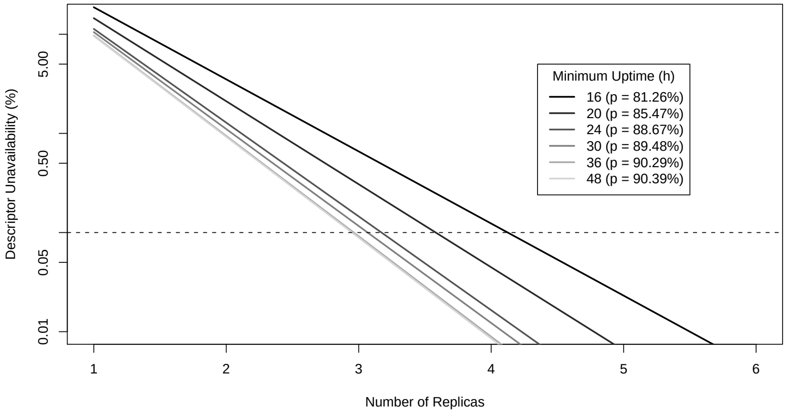


Figure 4.11: Descriptor unavailability as a function of number of replicas

cas also proportionally influences the number of messages and circuits that a hidden server needs to build. Under the assumption that replicas are stored under completely independent identifiers, the overall descriptor availability a is a function of the descriptor availability of a single replica p and the number of stored replicas r: $a = 1 - (1 - p)^r$. Likewise, descriptor *un*availability u can be calculated as: $u = (1 - p)^r$. The previously found values for descriptor availability can be inserted for the descriptor availability of a single replica p; the 0.001-quantiles of the empirical distributions are used here with the rationale that obtained results hold for 99.9% of all empirical cases. Figure 4.11 contains plots of descriptor unavailabilities as functions of minimum uptime and number of replicas. The dashed line visualizes a targeted overall descriptor unavailability of 0.1%. For a minimum uptime of 24 hours, this value is reached at a replication rate of 3.17, rounded up to 4.

To summarize, the evaluation has shown that a minimum relay uptime of 24 hours reduces the directory size to 677 nodes in the mean as compared to a total number of 1,423 nodes in the network. In return, the remaining nodes exhibit relatively low churn rates of around 1.1%. The percentage of descriptors that a client can possibly fetch within a period of one hour is 94.8% in the mean. As a result, a number of 4 replicas are required to achieve an overall descriptor availability of over 99.9%.

4.7 Implementation

The proposed design has been implemented and deployed in the public Tor network. Most of the implementation work has been performed as part of the Google Summer of Code 2007 program between April and August 2007. The design changes have been described in more detail and discussed with the Tor developers in two Tor proposals [36, 37]. Integration into the Tor source code took place between September 2007 and January 2008 with the first fully functional implementation being released on November 10, 2007. Since then, multiple minor improvements and bugfixes have been included continuously. At the time of writing on September 30, 2008, there are 37 directory nodes and approximatively 734 hidden services using the new design. Two months later, on November 30, 2008, the number of directory nodes has increased to 84.

This section contains an overview of the implementation without going into the details of the source code. Tables A.1 and A.2 on pages 195 and following contain lists of all patches that have been included to add new features or fix bugs in the context of this implementation.

The following description contains the file formats and message contents that are exchanged to compile a hidden service directory list and to publish and fetch a hidden service descriptor. All presented formats are examples which are specified in more detail in the updated specification documents [63, 65]. The section concludes with brief evaluations of directory usage that were performed in September and November 2008, one year after including the design in the Tor source code.

```
Nickname gabelmoo
Address 88.198.7.215
ORPort 443
SocksPort 0
DirPort 80
HidServDirectoryV2 1
ContactInfo 1024D/F7C11265 Karsten Loesing <karsten.loesing@gmx.net>
ExitPolicy reject *:*
// ...
```

Figure 4.12: Configuration file of a hidden service directory node

Compiling the Hidden Service Directory List. Tor relays that should take part in the distributed directory need to set the configuration option `HidServDirectoryV2 1` in their configuration file. Until Tor version 0.2.1.6-alpha that was released on September 30, 2008, this option has a default value of `0`, so that relay operators need to explicitly activate this functionality. Later releases have a changed default value of `1`. Figure 4.12 contains an example configuration of a relay that will be used throughout the rest of this section.

Relays that are configured as hidden service directory nodes add this information to their router descriptors that they upload to the directory authorities. They do this even if their uptime is smaller than the minimum uptime of 24 hours which is ensured by the directory authorities. Figure 4.13 shows a router descriptor that the relay uploaded on March 24, 2008, 8:51 UTC. Here, the publication time does not coincide with the startup time; the real uptime of the relay at publication time of the descriptor is 1,709,794 seconds, which means that the last restart took place on March 4, 2008. The line `opt hidden-service-dir` indicates that the relay is ready to store and serve hidden service descriptors. The fingerprint `6833...` is equal to the node identity which is later used to determine responsibility for storing descriptors.

The directory authorities memorize the router descriptors and include

```
router gabelmoo 88.198.7.215 443 0 80
platform Tor 0.2.1.0-alpha-dev (r13792) on Linux x86_64
opt protocols Link 1 Circuit 1
published 2008-03-24 08:51:11
opt fingerprint 6833 3D07 61BC F397 A587 A0C0 B963 E4A9 E99E C4D3
uptime 1709794
bandwidth 2097152 2097152 1375382
opt extra-info-digest 727BB1FF2C7848D68592E2A3B482B9B82CEE3147
opt caches-extra-info
onion-key
-----BEGIN RSA PUBLIC KEY-----
// ...
-----END RSA PUBLIC KEY-----
signing-key
-----BEGIN RSA PUBLIC KEY-----
// ...
-----END RSA PUBLIC KEY-----
opt hidden-service-dir
contact 1024D/F7C11265 Karsten Loesing <karsten.loesing@gmx.net>
reject *:*
router-signature
-----BEGIN SIGNATURE-----
// ...
-----END SIGNATURE-----
```

Figure 4.13: Server descriptor of a hidden service directory node

```
network-status-version 3
vote-status consensus
consensus-method 2
valid-after 2008-03-24 12:00:00
fresh-until 2008-03-24 13:00:00
valid-until 2008-03-24 15:00:00
voting-delay 300 300
// ...
r gabelmoo aDM9B2G885elh6DAuWPkqemexNM YIOUhGiUCJmf18zoxWexVvV7sbg
    2008-03-24 08:51:11 88.198.7.215 443 80
s Authority Fast Guard HSDir Named Running Stable V2Dir Valid
v Tor 0.2.1.0-alpha-dev (r13792)
// ...
```

Figure 4.14: Extract from network status consensus

summaries of them in the network status documents which they vote on every hour. The authorities further try to maintain connections to all relays to ensure the minimum uptime of 24 hours. Only those relays which are configured as hidden service directories and which are available for at least 24 hours are assigned the HSDir flag in the network status votes and consensus. Figure 4.14 shows the network status consensus from March 24, 2008, 12:00 UTC that contains an entry for the relay that was mentioned above. The line beginning with r contains the base64-encoded [29] node identity aDM9.... The line beginning with s contains the flags that the authorities assigned to the node, including the HSDir flag.

Publishing and Fetching Descriptors. Hidden services can be configured to have their descriptors published on the central directory servers, in the distributed directory, or both. The current default at the time of writing this thesis is publication to both directories in parallel. Users can change this by setting HiddenServiceVersion to either 0 for the central directory or 2 for the distributed directory, overriding the default value 0,2. At some time in the future this default value might be changed to 2.

Before publishing a descriptor, a hidden server first needs to determine the current descriptor identifiers. Figure 4.15 shows an example descriptor that was published on March 24, 2008, 12:35 UTC. The identity of the hidden service is 05620F93F1A6C9E64686 which is the hash of the permanent key and which leads to the onion address avra7e7ru3e6mrug.onion. Both date and service identity are used to calculate the current time period 0000368A. The shown descriptor is the second replica, thus having a replica index of 01. An application of the SHA-1 hash function on time period and replica index leads to the secret identifier part D107... which is also included in the descriptor as base32-encoded [29] value 2ed5.... In the last step, service identity and secret identifier part are concatenated and put into a hash function. The result is the descriptor identifier 662D... which is contained in the first line of the descriptor as mywy..., encoded in base32. The descriptor is ready for upload to the responsible hidden service directory, which is the node as shown above with node identity 6833....

When a client attempts to establish a connection to the hidden server, it first needs to fetch its descriptor. The current default is to fetch descriptors from both the central directory servers and from the distributed directory. There is no way to configure this behavior, but at some time in the future the code might change to fetch only descriptors from the distributed directory. The steps to determine the current descriptor identifier are similar to those when publishing a descriptor. A client would fetch the descriptor in Figure 4.15 by sending a fetch request for descriptor identifier mywy... to the directory with node identity 6833....

Directory Statistics. The following statistics indicate the dissemination of the distributed directory in the public Tor network. First, Figure 4.16 shows the development of the number of directory nodes beginning from public deployment on November 12, 2007. Since then the distributed directory was used in a beta state for about 10 months without being widely

```
rendezvous-service-descriptor mywyh2hxyvytkol64piqzub2hxdznt4o
version 2
permanent-key
-----BEGIN RSA PUBLIC KEY-----
// ...
-----END RSA PUBLIC KEY-----
secret-id-part 2ed5uk7uh2sq7t5kkhws6hqt2mlumo4p
publication-time 2008-03-24 13:35:37
protocol-versions 2,3
introduction-points
-----BEGIN MESSAGE-----
// ...
-----END MESSAGE-----
signature
-----BEGIN SIGNATURE-----
// ...
-----END SIGNATURE-----
```

Figure 4.15: Version 2 hidden service descriptor

publicized. On September 11, 2008, a discussion on the users' mailing list led a number of Tor relay operators to change their node configuration and enable directory node functionality on their relays. On September 30, 2008, the first development version has been released that makes relays act as hidden service directory nodes by default and does not require to open the directory port in order to do so anymore. On November 30, 2008, 84 of 1,191 relays are running as hidden service directory nodes. It is expected that the number of directory nodes grows steadily and eventually approaches the total number of relays as assumed in the previous evaluation.

The second statistic in Figure 4.17 considers the number of requests that were recorded on a single hidden service directory node between September 25 and October 1, 2008. In this period, there were 37 directory nodes in the network in the mean. This comparably large number of directory nodes makes it difficult to extrapolate the number of requests to

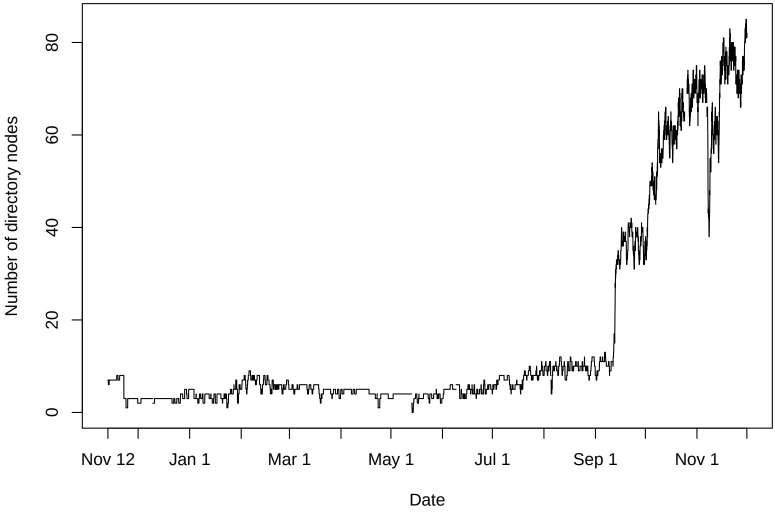


Figure 4.16: Development of directory nodes from November 12, 2007 to November 30, 2008

the distributed directory as a whole. For publish requests, there are two differences in the implementation as compared to the description here which are subject to change but need to be considered for this statistics: the replication rate is 6 instead of 4, and descriptors are re-published every hour. From this follows that the total number of hidden services being published in the distributed directory is roughly 6 times the number of publish requests to this node which are 734 hidden services in the mean. For fetch requests it is even harder, if not impossible, to estimate the overall number of requests to the distributed directory because of clients retrying failed requests.

4.8 Conclusion

To conclude, this chapter presented a novel distributed directory design to realize long-term recipient pseudonyms. The main requirements were

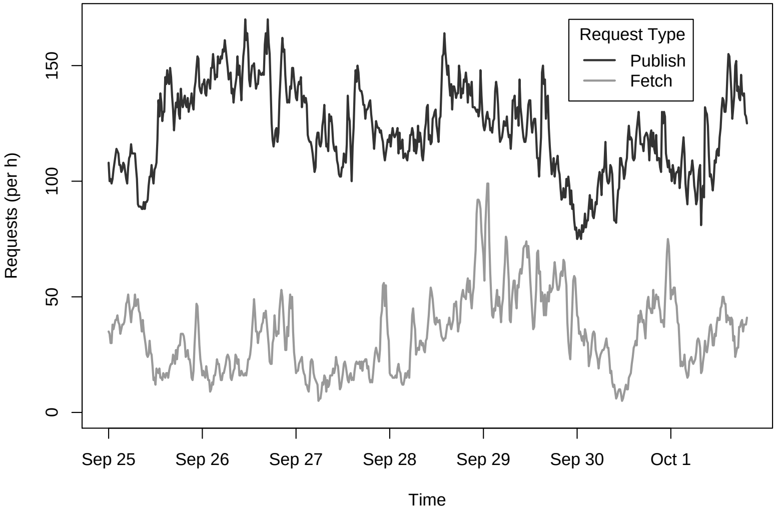


Figure 4.17: Requests processed by one directory node (per hour) between September 25 and October 1, 2008

to provide high availability and scalability, security properties like authenticity of entries, censorship resistance, and concealing activity, as well as support for private entries. A comparison of previous work has shown that earlier designs only supported subsets of these properties. The same also applies to the existing design of a hidden service directory in Tor which relies on a small number of central servers. The proposed design works by distributing the directory among hundreds of Tor relays. The design bears close resemblance to the distributed hash table Chord [76], although routing and replication have been adjusted to better meet the stated security properties. Possible security vulnerabilities have been discussed including measures to counter some of them. An evaluation based on archived network statuses and descriptors has shown the theoretical size of the distributed directory. This evaluation has also led to useful parameters for minimum uptime requirements to directory nodes and replication level of descriptors. The proposed design has been implemented and deployed in the public Tor network. It might replace the centralized design of the hidden service directory at some point in the future. Even though the proposed design is fitted to the requirements of Tor, the basic principles should be applicable to other pseudonymous service designs as well.

5 Client Authorization

The main focus of pseudonymous services is to hide the location of the service provider. In addition to that some, but not all, designs ensure authenticity of the service provider using a certain pseudonym towards the client. But so far no design has ensured authenticity and authorization of the *client* accessing a pseudonymous service. It may sound contradictory to the purpose of anonymous communication systems that clients could become linkable due to their using of authorization credentials. Performing client authentication and authorization reduces the protection of clients from anonymity to pseudonymity towards the service which is certainly not desired for the majority of services. But there are applications where it is advantageous to have client authorization towards a service, the private services that are motivated in this thesis being only one example. Possible benefits are improved denial-of-service protection, fine-grained access control, and individual quality of service.

It would be possible to simply add client authorization *after* establishing a connection to a pseudonymous service, but this approach has disadvantages: non-authorized clients could attempt to access a pseudonymous service, or they could illegitimately collect information about its activity or usage. It is therefore preferable that a service that performs client authorization reveals only as much information as necessary for authorized clients to access it. The main idea here is to perform client authorization already as part of the connection establishment process. The result is that such a service would not only be location-hidden, but also literally invisible to non-authorized clients.

There are few mentions of client authorization for pseudonymous services in previous work. The Tor design paper [11] and the Tor specification documents [65] mention client authorization for Tor hidden services. However, a working design has never been implemented in Tor. Øverlier and Syverson [57] revisit the topic and give a few hints of how client authorization could be performed, but do not describe a working design, either. An earlier version of the design proposed here has been presented on HotPETs 2008 [45].

This chapter is structured as follows: The next section discusses desirable properties of a pseudonymous service performing client authorization. Section 5.2 compares two different approaches for adding client authorization as a separate layer on top of an established connection to a pseudonymous service and shows the problems of these approaches. Section 5.3 proposes a basic extension of the Tor hidden service protocol to support client authorization, which is efficient, but which does not attempt to hide service activity and usage. Section 5.4 further extends this design to also hide service activity and usage. Section 5.5 presents an analysis of possible security properties of the new approach. Section 5.6 sketches some facts of the implementation. Section 5.7 concludes.

5.1 Requirements

There are some basic requirements to pseudonymous services with client authorization: At first, the original goal to hide the location of the server remains unchanged. Further, the server shall perform authorization, so that non-authorized clients are prevented from accessing a service. Besides these basic requirements, a pseudonymous service that performs client authorization should also fulfill two more specific requirements:

The first requirement is to *prevent unauthorized access attempts.* Access attempts performed by non-authorized users should be blocked *as early as possible.* These access attempts do not only waste resources, but could also be used to perform an attack on the server. Øverlier and Syverson took advantage of the fact that a server answers all client requests to reveal its location [56]. Further, swamping a pseudonymous server with client requests might be an effective attack to make the service unavailable. The best protection against these attacks is to prevent connection establishment in the first place.

The second requirement is to *conceal service activity and usage*. Non-authorized users must not be able to learn about service activity. In some cases service activity is consistent with the service provider's online activity. An adversary could exploit knowledge about service activity over time to learn useful information like timezone or personal online behavior of the service provider. One particular requirement here is simplicity to remove authorization from clients. In such a case, a service that once permitted access to a user should appear unavailable once authorization for that user is removed. Being able to remove a user without trace is useful for temporarily granting access to a service. Besides concealing service activity, service usage shall also be protected. Nobody except the service provider should be able to generate profiles of client requests to a service, even if requests cannot be attributed to specific clients.

5.2 Existing Client Authorization Approaches

In principle, client authorization can be performed with every pseudonymous service design. Every potentially authorized client would establish a connection to the pseudonymous service and run an authorization protocol. If the client succeeds, it is allowed to send and receive application data. If the client fails, the connection is torn down.

It is important to notice that a pseudonymous server in such a scenario needs to actually run a subsequent protocol to authorize clients. The service provider should not rely on the fact that the service is hidden, so that only authorized clients can connect in the first place. In practice, many service providers do not require client authorization if they deem the service address to be known to trusted clients only. Guha and Francis [26] have found that most private HTTP servers using dynamic DNS addresses do not perform subsequent authorization, but rely on the mere fact that their hostname is not leaked. But the authors used several techniques to enumerate hostnames of dynamic DNS providers, hence proving the original assumption false.

Another important assumption is that the authorization process itself does not leak information about the service. Not-yet-authorized clients should not gain any information from a failed authorization attempt about the nature or contents of a service. These information include service name or description as part of the authorization protocol. Otherwise, an adversary could utilize this information to find possible targets for subsequent attacks. The assumption for the following discussion is that services use a unified interface for their authorization protocol. As a result, an adversary would not have an incentive to attack one of these services without having further information. Such an attack would be untargeted against all possible services in the system, which is not a useful threat to defend against.

There are (at least) two different approaches to implement pseudonymous services with subsequent client authorization: In the first approach, a service provider sets up a single service for all users and performs user authorization after successful connection establishment. A second approach is to configure a separate service for each client in addition to a later authorization when the connection is established. In the following, these two approaches are discussed in more detail together with the possible security issues they raise.

5.2.1 Single Service for All Users

In the first approach a service provider sets up a single pseudonymous service for all users and performs user authorization after the connection is established. All clients use the same service address, but each client has separate authorization credentials for the subsequent authorization protocol. Whenever the service provider wants to grant access to a new user, he hands out his service address and new credentials. The other way round, when the service provider wants to remove authorization for a specific client, he removes the credentials from the list of permitted clients.

There are a number of security problems in this approach: An adversary can perform *unauthorized access attempts* by opening an unrestricted number of connections to the service and attempting to access it before failing at subsequent client authorization. With every established connection a non-authorized client might waste resources of the server that are used for connection establishment. This attack can be performed by all former clients that were removed from the list of authorized clients but who still know the service address.

Former clients can also *track service activity* even if they are not authorized to access the service anymore. The easiest way to do so is to periodically try to establish a connection to the service and conclude that a service is active if they succeed. It may also be sufficient to periodically request contact information and derive activity from an included timestamp or the mere fact that the contact information is updated.

Finally, a former client could *monitor anonymous service usage* when observing connection establishment to the known service address. In most systems this requires controlling part of the network infrastructure, which is harder than tracking service activity, but not impossible.

5.2.2 Separate Service For Each Client

An alternative approach for client authorization for pseudonymous services is to establish a separate service for each client. In this setting every client connects to a distinct service address and presents its credentials there. The service provider hands out a new service address *and* authorization credentials to a new user. In order to remove authorization, the service provider stops providing the corresponding service.

From this follows that the attacks as described for the first approach do not apply. The only entities that can link a service address to a service provider are the service and one client. As soon as the service is stopped, the removed client cannot learn anything about the service provider, because she does not know the service addresses of other clients accessing the service.

However, the effort that is required to set up a separate service for each client can be tremendous. The costs for maintaining contact points and pseudonyms grows linearly with the number of clients. These costs limit the number of clients that can reasonably be authorized and puts significant load on the network.

Further, a new problem arises: A service provider who offers more than one service usually advertises them in the network at the same time. These events give hints to the network, that two or more services might be run by the same provider.[24] An authorized client could learn about this relation by linking its own pseudonym to others. Even when authorization for a client is removed, the client can still attack or profile the service by using one of the other service addresses.

5.2.3 Separate Service For Groups of Clients

Between the two extremes of using a single service for all users and a separate service for every user, one could also offer a pseudonymous service for a group of users. Øverlier and Syverson [57] proposed grouping users and permit access to a service based on the group a user is in. Group-

[24] In case of Tor hidden services, the hidden server could pick the same introduction point or store hidden service descriptors on the same directory node for more than one service in quick succession.

ing users helps balancing out the advantages and disadvantages of both approaches. However, such a hybrid approach is not discussed further, because it does not add any new aspects besides of those of the two previous approaches.

5.2.4 Conclusions from Existing Approaches

The previous discussion has revealed inherent weaknesses of performing client authorization *after* establishing a connection to a pseudonymous service. As an alternative, client authorization could already be performed *during* connection establishment. The result is that non-authorized clients do not learn as much information about a service. The modifications to include client authorization in the connection establishment process are demonstrated with the Tor hidden service protocol [11]. Some of the changes are specific to Tor hidden services, but the design principles behind them could be adapted to other pseudonymous communication systems as well.

Two modified hidden service protocols are described in the following: First, a *basic protocol* is presented that has the goal to fulfill the security requirements as good as possible while using as few additional resources as necessary. Second, a *stealth protocol* is discussed that aims at providing the maximum possible security properties even at the expense of limited scalability.

5.3 Basic Pseudonymous Client Authorization Protocol

The main goal of the first client authorization protocol for Tor hidden services, the basic protocol, is to deny access to unauthorized clients, including introduction points and directory nodes as neutral network components. Unauthorized clients shall not be able to establish a connection to a hidden server and start exchanging application data. As a further requirement, unauthorized access attempts shall be blocked as early as possible. The hidden server should not be forced to use limited resources for unauthorized clients, especially with regard to building or extending circuits. Another requirement that can only be fulfilled to a certain extent is to limit propagation of service activity and usage in the network, that is, to introduction points and directory nodes. The fewer entities can observe actions related to a certain service identity, the fewer can create profiles of service activity or usage.

5.3.1 New Introduction Key for Introduction Points

The first modification is to reduce the information that an introduction point learns about a hidden service. An introduction point is not required to learn *the* identity of a hidden service, but only *an ephemeral* identity created by the hidden service to perform its task, which is to match incoming client requests with a previously registered hidden service. A client needs to learn about that identity before contacting the introduction point and be assured that it belongs to the hidden service. But there is no need for the identity that an introduction point learns to be equal to the service identity or even to persist for longer than a single introduction point establishment. Using a one-time identity for establishing an introduction point prevents the relay from recognizing a hidden service. The result is that even an introduction point cannot fetch the current descriptor of the hidden service and make an anonymous access attempt using one of the other introduction points.

When establishing an introduction point, the hidden server creates a new asymmetric *introduction key* and sends the public key to the introduction point instead of the public service key in the ESTABLISHINTRO cell. The implementation of an introduction point does not need to be modified for this change. In fact, an introduction point cannot tell an introduction key and a service key apart and might assume that it works on behalf of continuously changing hidden services.

The hidden server includes the public introduction keys of all established introduction points in the hidden service descriptor that it publishes to the directory nodes. The client learns about the introduction points and corresponding introduction keys from the directory and can be sure that the introduction point works on behalf of the service that it wants to connect to from the descriptor signature. The client uses the introduction key to create the INTRODUCE1 cell that it sends to the introduction point. Again, the introduction point does not notice that it is an introduction key and no service key, so that its behavior during introduction remains the same. Finally, the hidden server uses the introduction key instead of its permanent key to decrypt introduction requests in received INTRODUCE2 cells.

5.3.2 Encryption of Introduction Points in Hidden Service Descriptor

The second modification to the hidden service protocol is to encrypt the list of introduction points in hidden service descriptors. The intention is to prevent the directory nodes and non-authorized clients from establishing a connection to a hidden server. Without knowing which relays the hidden server uses as introduction points for a hidden service, an unauthorized client cannot send an introduction request to the hidden server. Even when guessing which relays might work as introduction point for the service or by flooding the network, a client that is not aware of the introduction key cannot create a valid INTRODUCE1 cell that gets passed by the introduction point.

While it may be appealing to encrypt the complete descriptor and not only the list of introduction points, this cannot be done here. The directory node still needs to understand certain parts of a descriptor to be able to verify legitimacy of storing the descriptor under the claimed identifier. These indispensable parts include the public service key and the secret identifier part as introduced in the last chapter. If these parts were encrypted, too, an adversary could attempt to store an arbitrary descriptor under a given descriptor identifier without the directory node being able to detect it. As discussed in Section 4.4.3, this would make the hidden service unavailable to clients: In case that only one descriptor is stored, clients would have no way to reach the legitimate hidden service anymore. In case of storing all descriptors on the directory, clients would have to download them all to filter out the legitimate ones, which is a race that clients would inevitably lose.

The task of encrypting introduction points needs to fulfill certain requirements: The storage overhead for encrypting the introduction points in a descriptor for a possibly large number of clients should be as small as possible. In addition to this, nobody should be able to tell the exact number of or detect changes to the set of authorized clients.

The protocol as described here makes use of symmetric keys which are called *client cookies* in the following. A hidden server generates client cookies for its clients and distributes them outside of Tor. When generating a hidden service descriptor, the service starts with encrypting the introduction-point part with a randomly generated symmetric session key. A symmetric block cipher is used here, more precisely AES [54] in counter mode [14] with a randomly generated initialization vector. Afterwards, the service encrypts the session key for all authorized clients using their client cookies, again with AES. The service generates *client identifiers* for all authorized clients by applying a hash function on the concatenation of client cookie and the initialization vector that has been used for encryption. The first four bytes of a client identifier are prepended to the encrypted session key, so that clients can efficiently search for the session key that was encrypted for them. In the rare case that two or more client identifiers are equal, clients need to decrypt all session keys that could have been encrypted for them and try to decrypt and parse the introduction points with them. If the number of authorized clients is not a multiple of 16, the service adds fake entries consisting of random numbers to conceal

Table 5.1: Encryption of introduction points for basic client authorization protocol

Field	Description
Client blocks	Number of clients divided by 16, rounded up to next integer
For each client	
Client identifier	Hash of client cookie and initialization vector
Session key	Session key encrypted with client cookie
End of client part	
Padding	Random data to fill client identifiers and session keys for clients up to next multiple of 16
Initialization vector	Initialization vector for AES encryption
Introduction Points	List of introduction points encrypted with session key

the exact number of clients. Encrypted session keys are ordered by client identifiers in order to conceal addition or removal of authorized clients. Table 5.1 shows the encryption scheme for introduction points.

5.3.3 Client Authorization at Hidden Server

The last change to the hidden service protocol is authorization at the hidden server. Even though encryption of introduction points in hidden service descriptors ensures that only authorized clients learn about the introduction points and corresponding introduction keys, a removed client might still access the hidden service until all established introduction points have changed. Further, the introduction points could attempt to access the service if no further authorization was required at the hidden server by generating an INTRODUCE2 cell on their own. As a side effect, authorization at the hidden server enables the hidden server to selectively remove authorization of a client. When clients identify themselves at a hidden server using the credentials they obtained before, the hidden server can attribute possible misuse to one of its clients and remove authorization. If the hidden server would not be able to do so, a single rogue authorized client could make the service useless for everyone.

The client includes its client cookie in the encrypted introduction request that it sends to the introduction point in the INTRODUCE1 cell and that is forwarded to the hidden server in the INTRODUCE2 cell. The hidden server checks whether a contained client cookie is valid before extending a circuit to the specified rendezvous point.

The service also needs to prevent replay attacks performed by rogue introduction points. An introduction point could attempt to replay valid introduction requests to force the hidden server to repeatedly extend new circuits to the enclosed rendezvous point. While this would not allow the introduction point to circumvent authorization, it would waste resources of the service and could be used to track down the location of the hidden server [56]. As a defense, the hidden server memorizes the first part of the Diffie-Hellman [10] handshake of valid requests and drops duplicates. The service requires that the timestamp of an INTRODUCE2 cell is no more than 30 minutes in the past or future and that the first part of the Diffie-Hellman handshake has not been used in the past 60 minutes.

5.3.4 Summary of Basic Pseudonymous Client Authorization Protocol

The basic client authorization protocol changes the messages involving introduction points and directory nodes, namely ESTABLISHINTRO, INTRODUCE1, and INTRODUCE2 cells as well as the hidden service descriptor format. Introduction points do not learn the identity of a hidden service as a result of using freshly created introduction keys. Directory nodes and unauthorized clients do not learn the set of introduction points that is encrypted for authorized clients only. Introduction points are not allowed to access the hidden server over the introduction circuit themselves. Only authorized clients are allowed to establish a connection to a hidden server. The protocol change does not require adding any new protocol steps, but only changing the contents of exchanged messages.

5.4 Stealth Pseudonymous Client Authorization Protocol

The basic authorization protocol exhibits better security properties than existing solutions that perform client authorization after connection establishment. However, the common service descriptor that is shared by all authorized clients still reveals too much information for some services. Whoever knows the identity of a given hidden service can periodically request the service descriptor from the directory and conclude service activity from recent descriptor updates. Further, a directory node that observes fetch requests for a given service descriptor can generate usage patterns for the hidden service.

The stealth client authorization protocol for hidden services aims at concealing service activity and usage from unauthorized entities. Similarly to the second approach of performing client authorization after connection establishment, access to hidden services is issued on a per-user basis. Every authorized client knows a separate service identity to access the service. When the server decides to remove authorization of a client, it stops publishing descriptors for that identity. The removed client cannot attribute observed descriptor requests to service usage, because the other clients use different service identities to access the service.

The main drawback of the stealth protocol is its limited scalability.[25] On the one hand, the same introduction parts can be reused for all authorized clients, thus keeping the number of concurrently open circuits constant. But on the other hand, publishing distinct service descriptors produces significant additional network load. This requirement limits the number of clients that a hidden server can authorize using the stealth

[25] Applying a hybrid approach with grouping users could mitigate these scalability problems. In such an approach the hidden server would publish separate descriptors for the groups and perform authorization based on client-specific authorization data.

protocol. Special precautions need to be taken to hide relations between service identities when publishing hidden service descriptors. These precautions include storing descriptors on distinct directory nodes and with random delays, so that linking of descriptors is at least made more difficult for an adversary.

5.4.1 Client-specific Service Identities

The first design change of the stealth authorization protocol is to generate client-specific hidden service descriptors. The descriptors for different clients of a service shall not be linkable to each other. Therefore, a hidden server creates a separate asymmetric *client key* for each client replacing the service key that is used to generate a hidden service descriptor. As a result, all published descriptors seem to originate from distinct services. With the distributed directory as described in the previous chapter, descriptors with different identities are likely to be stored on distinct directory nodes.

Using distinct identities for authorized clients of a service is an important first step, but it does not suffice to hide links between identities. At least three problems remain: The first problem is that a directory node that once learns about an identity can subsequently track activity of that identity by attempting to download descriptors published for the identity. As a result, the directory node can detect if different descriptors are published roughly at the same time and link them. The second problem is that all descriptors published by one service contain the same set of introduction points. While this helps improving scalability of the approach, it also reveals links between descriptors. The third problem is simultaneous publication of two or more descriptors to the same directory node or to collaborating directory nodes. When observing such a simultaneous publication, an adversary can link the identities in the future. Solutions for these three problems are presented in the following.

5.4.2 Private Entries in Descriptor Directory

The fact that the descriptor is only downloaded by a single client allows to make use of private entries as described in Section 4.4.3. Both service and client can share a secret *descriptor cookie* that is required to generate descriptor identifiers. As a result, directory nodes that learn about the identity of a hidden service descriptor are not able to download future descriptors or find other replicas of the descriptor. Further, they cannot generate a new node identity key and hope to become responsible for the descriptor in the future. This property is ensured by the secure hash function that is used to generate the secret identifier part. The one-way property of the hash function prevents the directory node from extracting the descriptor cookie from the secret identifier part. But without knowing the descriptor cookie, neither a directory node nor any other entity can determine the next secret identifier part and thereby future descriptor identifiers.

```
descriptor-id = H(public-key-id || secret-id-part)
secret-id-part = H(descriptor-cookie || time-period || replica-index)
```

Even though descriptor identifiers change in an unpredictable way, a storing directory node can still verify the legitimacy of a descriptor to be stored under a given identifier. Therefore, the directory node verifies the descriptor signature and attempts to generate the descriptor identifier using the contained secret identifier part. An adversary with a different public key would be unable to find a secret identifier part that results in the same descriptor identifier.

5.4.3 Encryption of Introduction Points

All service descriptors that a service publishes for its clients contain the same set of introduction points. Sharing introduction points for multiple clients significantly reduces the efforts to set up a service with stealth client authorization. However, the fact that multiple service descriptors contain the same set of introduction points and introduction keys would reveal a link between them. Therefore, the introduction point part needs to be encrypted for the client. Encryption the introduction points also prevents the directory nodes from attempting to access a service themselves. As opposed to the basic client authorization protocol, the list of introduction points only needs to be encrypted for a single client. The symmetric descriptor cookie, which is also used to generate the descriptor identifier, is used as encryption key here.

5.4.4 Delayed Descriptor Publication

With the changes so far it is *almost* impossible to link two or more hidden service descriptors for different clients to be issued for the same hidden service. The only problem that remains is simultaneous publishing of two or more descriptors to the same directory node or to collaborating directory nodes. For one thing, they contain the same or very close timestamps. For another thing, the upload requests arrive in short order.

Descriptor uploads are performed in one or more *upload rounds* with each of them lasting for one *upload period* of, for example, 30 seconds. The hidden server prepares the descriptors for all clients to determine to which directory nodes they would be uploaded. The service ensures that at most one descriptor is uploaded to any directory node in the current upload round; the other descriptors are postponed to the next round. When uploading descriptors, the hidden server does not send upload requests all at once, but distributes upload times over the upload period at random. After an upload round is complete, the service starts over with the next one, considering all those descriptors that have not been uploaded yet.

Certainly, this does not prevent linking entirely, but it makes it harder. There is a conflict between hiding links between clients and making a service available in a timely manner.

5.4.5 Summary

The stealth client authorization protocol extends the basic protocol by assigning each client a separate service identity to access the hidden service. The server publishes separate service descriptors for its clients that are generated using distinct client keys instead of the service key. While this reduces scalability to a certain extent, the same set of introduction points can be reused for all clients, so that the service only needs to keep a constant number of circuits open. The descriptors are stored as private entries in the descriptor directory using a previously shared descriptor cookie. As a result, directory nodes cannot track existence of a descriptor over time or try to become responsible for future descriptors of a service. The descriptor cookie is also used to encrypt the introduction points that are contained in the descriptor for the requesting client. The upload of descriptors for all clients is delayed for random times in order to hide relations between the identities that clients use to access the hidden service.

5.5 Security Analysis

At the beginning of the chapter it was found that the attempts to perform client authorization after connection establishment to a pseudonymous service reveal a number of security problems. These include unauthorized access attempts and tracking of service availability and usage. A security analysis shall evaluate whether the two proposed protocol changes improve these security problems and which problems do remain.

The security analysis comprises four different protocols: The first two are connection establishment to a single service or a separate service for each client combined with subsequent client authorization. In these protocols, the unchanged Tor hidden service protocol is used, that is, the protocol with server-based descriptor storage and without using freshly generated introduction keys [11].[26] The other two protocols are the basic and the stealth authorization protocol as described above.

An important part of a security analysis is to define the capabilities of the adversary. In the following, three classes of adversaries are distinguished depending on their current or past permissions to access a service: authorized clients, removed clients, and non-authorized clients. An authorized client has the permission to access the service under investigation at the time of evaluating security properties. A removed client was once authorized by the service under investigation, but the service has removed authorization by the time of the evaluation. A non-authorized client was never authorized to access the service, but has learned about the service somehow, for example, when acting as introduction point or directory node. This classification has at least two implications: The first implication is that only authorized and removed clients know about the provided contents or the type of a service and might have a direct motivation to attack the service; the non-authorized client only knows that the service exists. An adversary that is a non-authorized client collaborating with either an authorized or a removed client is classified as the latter type. The second implication is that a non-authorized client does not necessarily have less information than a removed client: if the service uses multiple identities, the non-authorized knows at least about an actively used service identity, while the removed client might only know the service identity that was assigned to it when it was authorized. The three adversary classes have in common that they all know a current or past service identity. Adversaries who do not know any hidden service identity are considered to be too limited to effectively mount an attack on hidden services as discussed here. They are therefore excluded from the analysis.

[26] The distributed directory and introduction keys also improve security properties of services that do not perform built-in client authorization. However, the goal of this analysis is to compare the protocol as it was deployed prior to this thesis with the proposed protocol changes.

Besides this classification, an adversary can have more knowledge about the service. If the server uses multiple identities for its clients, the adversary might know the identities that are used by other, currently authorized clients. Further, the adversary might have the capabilities to attack or control a limited number of relays to act as introduction point or directory node in the distributed hidden service directory.

The other part of performing a security analysis is to define possible targets of an adversary. These are the security properties that shall be protected. The following analysis comprises five possible attacks: accessing a service without permission, performing a denial-of-service attack, censoring a service, and tracking either service activity or client requests to a service. For each attack, the required steps by an adversary as well as possible protections by a service are discussed. Table 5.2 at the end of this section summarizes the attacks and shows how effective protections against them are.

5.5.1 Access Service without Permission

The basic protection that all protocols shall provide is to prevent clients without authorization from accessing the service, including both removed and non-authorized clients.

In the first two protocols, the subsequent authorization step ensures that unauthorized requests are not permitted. Only clients with current authorization data are able to pass this step. In the second protocol that uses separate services, a removed client further needs to know the service identity of another, currently authorized client to make an access attempt.

The basic and stealth protocols stop unauthorized access attempts even earlier. In the basic protocol an adversary fails at decrypting the introduction points and introduction keys that are contained in a service descriptor. Hence, the adversary would not know where to send the INTRODUCE1 cell and what public introduction key to use to generate it.

In the stealth protocol, an adversary would fail even one step prior to that, that is, when trying to download the service descriptor without knowing the descriptor cookie of an authorized client. Even if an adversary in the stealth protocol runs a directory node that is responsible for storing a service descriptor by chance, the introduction points are encrypted and unreadable for the adversary.

5.5.2 Perform Denial-of-Service Attack

Even though it is not possible to gain access to a service without the required authorization data, an adversary might want to mount a denial-of-service attack against that service. Therefore, the adversary sends a multitude of introduction requests, forcing the hidden server to build circuits to the contained rendezvous points. If the frequency of requests exceeds a certain level, the service is unable to answer legitimate requests in time or at all. A similar attack has been performed by Øverlier and Syverson [56] to reveal the location of a hidden server.

In the first protocol that performs client authorization after connection establishment to a single service, almost everyone who has once learned about the service identity can perform this attack. The server has no way of attributing the attack to one of its clients. In the second protocol that uses separate services for authorized clients this situation is only improved to a certain extent. The service can distinguish the service identity that is attacked and could attribute misuse to the client this identity was assigned to. However, the attack can also be performed by removed clients that have learned about the service identity that is used by another client or by arbitrary, non-authorized clients. From this follows that the conclusion to remove authorization for the presumably attacking client may be false. While the adversary would not achieve the original goal of making the service unavailable for all clients, single legitimate clients would be affected by the adversary's denunciation.

The two protocols proposed here exhibit better protection against denial-of-service attacks. If an authorized client would mount such an attack, the service could reliably attribute misuse to the client and safely remove authorization. Removed and non-authorized clients cannot perform the attack, because they cannot even get past one of the introduction points for the same reason as discussed for accessing a service without permission.

5.5.3 Censor Service

Another way of making a service unavailable is to disturb legitimate clients in their attempts to establish a connection to it. An adversary could try to make the service descriptor unavailable or control the introduction points and drop introduction requests.

In the case of a single service with subsequent client authorization, an adversary could attempt to force the central directory servers to stop serving descriptors of a given service. In the original protocol it is sufficient to prevent a single directory server from serving descriptors to achieve at least partial censorship. Another way to censor a single service is to run relays and wait to be picked by the service for establishing all introduction points on them. When clients send introduction requests to the service, the relays can refuse to forward them to the service. Neither of these two attacks would be noticed by a service.

Using separate services for different clients makes these two attacks significantly harder, if not impossible. An adversary would need to know all service identities to be censored. For one thing, the efforts that are required to force the directory servers to censor multiple descriptors might be slightly higher than to censor only one descriptor. For another thing, all services use a different set of introduction points, so that controlling all of them by chance is almost impossible.

The possible attacks in the basic authorization protocol are different. An adversary can attempt to control all directory nodes in the distributed directory that are responsible for the service descriptor. In this case the countermeasures of the distributed directory against censoring services as described in the previous chapter come into play. If the adversary is an authorized client, she can try to control all introduction points and block introduction requests there for the other clients. If the adversary is not authorized, she would not know whether her own relays have been picked as introduction points. She would not be able to decrypt the introduction points contained in the service descriptor and compare introduction keys.

An adversary being an authorized client has the same capabilities to censor a service in the stealth protocol as in the basic protocol. This property is the only one in which the stealth protocol is more vulnerable than the approach to set up separate services. The reason is that only a single set of introduction points is established as compared to a separate set for each client. In contrast to this, the protection against removed and non-authorized clients is by far better in the stealth protocol. Such an adversary cannot run her own relays and hope to be picked as introduction points, because she has no way to confirm that she works for the service. A minor exception is a recently removed client being the adversary: This adversary could attack the known introduction points to make the service unavailable. However, as soon as the hidden service has established new introduction points, it is available to the other clients again. Further, neither removed nor non-authorized client as adversary can guess positions of future service descriptor of other identities used by the service. Thus, no such adversary can control the responsible directory nodes on purpose, but only hope to become responsible by chance.

5.5.4 Track Service Activity

One of the main goals of improving client authorization for pseudonymous services was to prevent tracking of service activity. This goal does not apply to currently authorized clients who need this information in order to legitimately establish a connection. The focus is rather on removed clients who shall not be able to observe service activity after their authorization is removed. Non-authorized clients shall not be given the possibility to track service activity of a yet-unknown service, either, which they could later combine with other information they might gain about the service.

The first protocol based on a single service lacks any protection against tracking service activity. Any adversary who has once learned the service identity can periodically download the service descriptor. An update of the service descriptor indicates service activity. The adversary can also attempt to establish a connection to the service to confirm that it is active. Alternatively, the adversary can run own relays and wait for one of them to be picked as introduction point to track service activity.

The second protocol based on separate services is exposed to the same threats concerning service activity. As soon as the adversary knows at least one identity that the service uses, she can periodically download service descriptors and attempt to establish connections to them. Running own relays and waiting to be picked as introduction point works even better in this protocol, because the server needs to establish separate introduction points for all its clients.

The basic authorization protocol does not improve the protection of service activity much. The only difference to the first two protocols is that the adversary cannot establish a connection to the service to confirm activity, because introduction points are encrypted in the descriptor.

The stealth protocol, finally, does protect service activity. An adversary needs to know at least one currently used service identity and has to run own directory nodes. She then waits to become responsible for one of the service descriptors to observe descriptor publications. Similar to the basic protocol, the adversary cannot attempt to connect to the service to confirm activity. Even if an adversary can put significant resources on tracking service activity (and is willing to do so), the result is a rather coarse-grained activity profile with a number of missing intervals.

5.5.5 Track Client Requests

The last possible attack is tracking client requests. An adversary might be interested in knowing the frequency and pattern of requests to a given service. Adversaries include currently authorized clients as well as removed or non-authorized clients.

For the first protocol that performs client authorization subsequent to connection establishment, the adversary needs to control at least one of the introduction points of a service by chance. In this case the adversary learns about all requests that are performed using this introduction point. While these constitute only a certain share of all client requests, they suffice to obtain a general pattern. Another, less likely option is to force the directory server operators to reveal this information.

In the protocol based on separate service, the probability of running one of the introduction points increases. However, the adversary can only observe requests by one client with every controlled introduction point in this way.

In the basic protocol, an authorized client as adversary learns about client requests when controlling either introduction points or directory nodes. An adversary that is currently not authorized cannot make use of data collected at introduction points. She cannot know whether the introduction point acts on behalf of the service. Running a directory node is even more effective in this case, because the adversary can become responsible for future service descriptors on purpose.

The stealth protocol makes it harder to track client requests for removed and non-authorized clients, but not for authorized clients. An authorized client as adversary can run relays and wait to be picked as introduction point. With all clients using the same set of introduction points, the adversary can track client requests rather easily. Clients without current authorization need to run relays in hope to be picked as directory nodes for some of the identities used by the service. They can then learn from fetch requests for a descriptor that clients attempt to access the service. This attack works only for a small number of client requests for known identities used by the service.

5.5.6 Summary

In summary, the two proposed protocols exhibit better protection than the existing approaches to establish a connection to a service and perform authorization subsequently. Both existing approaches are vulnerable to denial-of-service attacks which is solved in the two new protocols. The stealth protocol protects service activity and usage to a certain extent, but not completely, so that an adversary with sufficient resources can still observe fragments of service activity and usage. Table 5.2 summarizes the results of the security analysis.

5.6 Implementation

The two proposed client authorization protocols as extensions of the hidden service protocol have been designed and implemented between April 2007 and September 2008. First, the prerequisites have been created during the Google Summer of Code 2007 program. These include introduction keys as a means to hide service identity from the introduction points and descriptor cookies for storing private entries in the distributed descriptor storage [36]. In September 2007, an infrastructure for client authorization in hidden services has been proposed. This proposal also includes the basic and stealth protocols as two instances of authorization protocols for hidden services [31]. The implementation work was mainly done in summer 2008 and added to the codebase between August and September 2008. Table A.3 contains a list of the patches. Both hidden servers and clients need to update to Tor version 0.2.1.6-alpha to make

Table 5.2: Evaluation of attacks on pseudonymous services performing client authorization and effectiveness of protections against these

	Single Service	Separate Service	Basic Authorization	Stealth Authorization
Unauthorized access				
Removed client	⊕⊕[1]	⊕⊕[1,2]	⊕⊕[3]	⊕⊕[2,3,4]
Non-authorized client	⊕⊕[1]	⊕⊕[1]	⊕⊕[3]	⊕⊕[3,4]
Denial-of-service attack				
Authorized client	⊖⊖[5]	⊖[6]	⊕⊕[7]	⊕⊕[7]
Removed client	⊖⊖[5]	⊖[2,6]	⊕⊕[3]	⊕⊕[2,3,4]
Non-authorized client	⊖⊖[5]	⊖[6]	⊕⊕[3]	⊕⊕[3,4]
Censor service				
Authorized client	⊕[8,9]	⊕⊕[8,9,10]	⊕[9,11]	⊕[9]
Removed client	⊕[8,9]	⊕⊕[2,8,9,10]	⊕[11]	⊕⊕[2,12]
Non-authorized client	⊕[8,9]	⊕⊕[8,9,10]	⊕[11]	⊕⊕[12]
Track service activity				
Removed client	⊖⊖[13,14,15]	⊖⊖[2,13,14,15]	⊖⊖[13,14]	⊕[2,4]
Non-authorized client	⊖⊖[13,14,15]	⊖⊖[13,14,15]	⊖⊖[13,14]	⊕[4]
Track client requests				
Authorized client	⊖[14]	⊖[14]	⊖⊖[14,16]	⊖[14]
Removed client	⊖[14]	⊖[2,14]	⊖⊖[16]	⊕[2,4]
Non-authorized client	⊖[14]	⊖[14]	⊖⊖[16]	⊕[4]

⊖⊖	trivial attack with almost no protection
⊖	realistic attack with only weak protection
⊕	realistic protection against an attack
⊕⊕	strong protection against an attack or not possible by design

Table 5.2 – continued from previous page

[1] Server is protected by subsequent authorization protocol.

[2] Adversary knows service identities of currently authorized clients.

[3] Server is protected by encryption of introduction points for authorized clients.

[4] Adversary controls one of the directory nodes that stores a service descriptor by chance.

[5] Server cannot attribute misuse to one of its clients.

[6] Server could attribute misuse to one of its clients and remove authorization which might be wrong.

[7] Server can reliably attribute misuse to one of its clients and remove authorization.

[8] Adversary forces directory server operators to censor service descriptor.

[9] Adversary controls all introduction points by chance.

[10] Server is protected by the fact that all clients use different set of introduction points.

[11] Adversary controls all directory nodes that store a service descriptor on purpose.

[12] Adversary controls all directory nodes that store a service descriptor by chance.

[13] Adversary periodically downloads service descriptor.

[14] Adversary controls at least one introduction point by chance.

[15] Adversary can attempt to establish connection to service.

[16] Adversary controls at least one directory node that stores a service descriptor on purpose.

use of the authorization protocols. In contrast to this, relays are not required to upgrade. This fact should significantly accelerate deployment of the new features.

The remainder of this section gives an overview of the implementation by showing examples of configurations and message formats. In the example, a user, Bob, provides a hidden service and grants access to two users, Alice and Carol. The example shows how Alice configures her Tor client and accesses the service. The use of both basic and stealth protocol are presented. Bob starts by configuring his Tor client to provide a hidden service with the basic client authorization protocol:

```
HiddenServiceDir /home/karsten/myhiddenservice
HiddenServicePort 80 127.0.0.1:8000
HiddenServiceAuthorizeClient basic alice, carol
```

This configuration instructs the Tor client to set up a hidden service with data directory `/home/karsten/myhiddenservice` that listens on the virtual port 80 and forwards incoming requests to the local service that runs on port 8000. The service generates authorization data for the basic authorization protocol for two users `alice` and `carol` and makes the service accessible for these authorized clients only. The service generates authorization data and stores them in a file `client_keys` with the following content:

```
client-name alice
descriptor-cookie L7DVicgw5z8Q85M4bloL0A==
client-name carol
descriptor-cookie YVXDZBHRh05qwmA5ql9X9g==
```

The service provider then can give out authorization data for his service to Alice and Carol. These data include the onion address and the client cookie (labeled descriptor cookie in the above file). The data are combined in another file `hostname`:

```
pnli33ynmrbt5p4l.onion L7DVicgw5z8Q85M4bloL0A # client: alice
pnli33ynmrbt5p4l.onion YVXDZBHRh05qwmA5ql9X9g # client: carol
```

Alice configures her Tor client to access Bob's service using the given authorization data. Therefore she adds a line to her configuration file:

```
HidServAuth pnli33ynmrbt5p4l.onion L7DVicgw5z8Q85M4bloL0A
```

Whenever she tries to access the specified onion address, her Tor client will use the given client cookie to decrypt the introduction points and send an introduction request to Bob's service.

If Bob wants to make use of the stealth authorization protocol, he needs to configure his service accordingly. He may only specify up to 16 clients in the stealth protocol as opposed to 512 in the basic authorization protocol:

```
HiddenServiceDir /home/karsten/myhiddenservice
HiddenServicePort 80 127.0.0.1:8000
HiddenServiceAuthorizeClient stealth alice, carol
```

The service generates client keys and descriptor cookies for all authorized clients and writes them to the file `client_keys`:

```
client-name alice
descriptor-cookie NLTXwrXuwHi4WRjAsZlyMg==
client-key
-----BEGIN RSA PRIVATE KEY-----
// ...
-----END RSA PRIVATE KEY-----
client-name carol
descriptor-cookie zIn16MC4qJnd939Jl5oTzA==
client-key
-----BEGIN RSA PRIVATE KEY-----
// ...
-----END RSA PRIVATE KEY-----
```

The service is then available for Alice and Carol under distinct onion addresses and descriptor cookies. These are written to the `hostname` file:

```
vh4t3oqu22umc5pm.onion NLTXwrXuwHi4WRjAsZlyMh # client: alice
6vstfml34qouxz5w.onion zIn16MC4qJnd939Jl5oTzB # client: carol
```

Alice configures her Tor client to use the authorization data when trying to access Bob's service in the same way as above. The difference is that the Tor client needs to include the descriptor cookie when trying to download the service descriptor which is not the case in the basic authorization protocol. The descriptor cookie contains the information that the authorization data is used for the stealth protocol as opposed to the basic protocol: The 22 characters-long base64-encoded [29] string may contain up to $22 \times 6 = 132$ bits of which 128 bits are used for the cookie. The remaining 4 bits are used to identify the authorization protocol. This is why the last character of the authorization string in the `hostname` file differs from the descriptor cookie in the `client_keys` file.

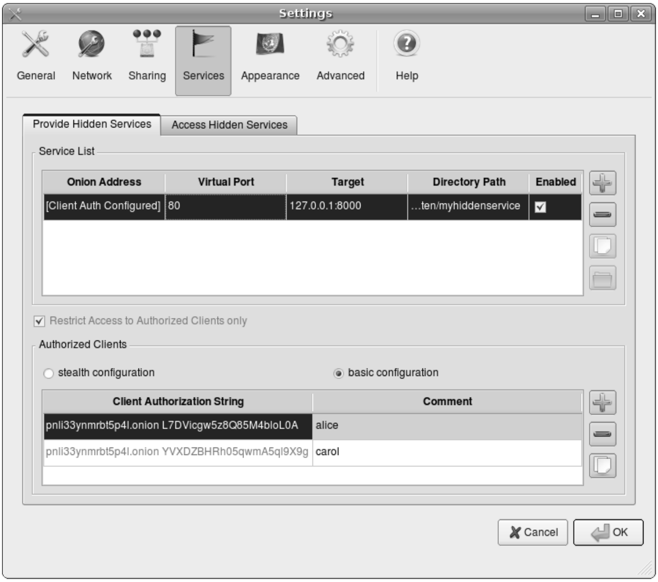
The ability to configure hidden services with client authorization has been added to the Tor controller Vidalia.[27] The necessary implementation work was performed by Domenik Bork in the Google Summer of Code 2008 program.[28] The motivation is to make it easier for users to provide and access hidden services with client authorization. Figure 5.1 shows the configuration of a hidden service with basic client authorization. Figure 5.2 shows how clients configure their Tor clients to use authorization data to access a hidden service.

5.7 Conclusion

This chapter has motivated the use of pseudonymous services with client authorization which are offered to a limited set of clients. An analysis of two approaches to use pseudonymous services with subsequent client authorization has shown several security problems that might turn out to be privacy-relevant for the service provider. Two extensions of the Tor hidden

[27] See the Vidalia homepage: `http://www.vidalia-project.net/` (last checked: Dec 17, 2008)

[28] See the accepted project application: `http://code.google.com/soc/2008/eff/appinfo.html?csaid=86500DD2D78BB5D9` (last checked: Dec 17, 2008)



Service List

Onion Address	Virtual Port	Target	Directory Path	Enabled
[Client Auth Configured]	80	127.0.0.1:8000	...ten/myhiddenservice	☑

☑ Restrict Access to Authorized Clients only

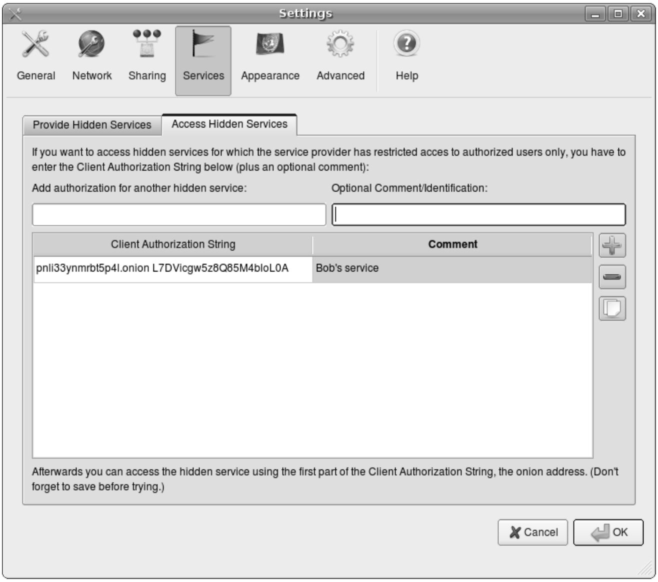
Authorized Clients

○ stealth configuration ● basic configuration

Client Authorization String	Comment
pnli33ynmrbt5p4l.onion L7DVicgw5z8Q85M4bloL0A	alice
pnli33ynmrbt5p4l.onion YVXDZBHRh05qwmA5ql9X9g	carol

Figure 5.1: Configuration of client authorization using Vidalia

service protocol have been presented that are designed to better support client authorization during the connection establishment process. A subsequent security analysis has shown possible attacks on the protocols and protections against them. The two new protocols have been implemented and deployed in the public Tor network.



If you want to access hidden services for which the service provider has restricted acces to authorized users only, you have to enter the Client Authorization String below (plus an optional comment):

Add authorization for another hidden service:

Optional Comment/Identification:

Client Authorization String	**Comment**
pnli33ynmrbt5p4l.onion L7DVicgw5z8Q85M4bloL0A	Bob's service

Afterwards you can access the hidden service using the first part of the Client Authorization String, the onion address. (Don't forget to save before trying.)

Figure 5.2: Configuration of access to hidden services using Vidalia

6 Performance of Pseudonymous Services

The performance of pseudonymous services is inherently worse than that of services that can be contacted directly. The first reason for this is that the multi-hop routing in anonymous communication systems adds delay and reduces bandwidth of relayed traffic. The second reason is that offering and accessing pseudonymous services require more steps than providing anonymity for clients accessing a public service. However, from a user perspective, pseudonymous services should meet minimum performance requirements which users can rely on. It is assumed that none of the delay in offering or accessing pseudonymous services is introduced by the system on purpose to defeat traffic analysis, but that all delay is the result of normal system operation.

Wendolsky and others [80] performed a comparative study of client-anonymous connections in Tor and in the AN.ON system [3]. They found that latencies of connections averaged to 4 seconds. They concluded from the studies by Köpsell [33] that these 4 seconds were the acceptable time that users are willing to wait: Whenever the number of users grows, the network load goes up and the average latency in the network increases, too. This bad performance deters less anonymity-aware users from using the system, so that the number of users decreases and the average latency improves again. This process stabilizes at a given number of users and corresponding average latency. While 4 seconds may sound high for the delay to connect to a service, pseudonymous services exhibit a delay that is a multiple of that. Performance problems may be considered the primary reason why pseudonymous services are not as popular as they could be.

There are a number of questions that shall be answered in the context of performance of pseudonymous services: How can the performance of pseudonymous services be measured? What is the performance that users experience in a deployed anonymous communication network? Which are the steps that are responsible for the bad performance? Is it possible to improve these steps in order to achieve a better overall performance of pseudonymous services?

The following analysis focuses on the performance of Tor hidden services. Most of this work has been conducted between May and December 2008 as part of the NLnet project to Speed Up Tor Hidden Services.[29] During this project a number of bugs in the Tor code could be fixed that were responsible for unnecessary delays of hidden services. Table A.4 on page 199 lists these bugfixes. Further, some changes to the hidden service protocol have been implemented and evaluated during this project [43]. Table A.5 on page 200 contains a list of these design changes. The analysis and the reasons for proposing these changes will be the subject of this chapter.

In the next section, a measurement setup is presented for performance measurements in the Tor network. Sections 6.2 and 6.3 cover the processes of making a hidden service available in the network and establishing a connection to it. They contain an analysis of performance of these tasks including several substeps and a discussion of possible design changes to improve performance. The challenge is to retain security properties while not to increase network load too much. Section 6.4 concludes the chapter.

6.1 Measurement Setup

The performance of Tor hidden services can best be measured by setting up and controlling a small number of Tor clients and relays that are con-

[29] See also the project page: `https://www.torproject.org/projects/hidserv.html` (last checked: Dec 17, 2008)

nected to the public Tor network. These nodes are then configured to participate in advertising a hidden service in the network or establishing a connection to it. In such a setting, one Tor client could be used to provide the hidden service and another one to request it. The Tor relays could act as introduction points, rendezvous points, or directory nodes; it is necessary, however, to change the Tor code to enforce using the own nodes for these tasks rather than randomly picked ones. It is neither required nor desirable to control all relays in circuits that either of the Tor clients builds. To the contrary, the obtained measurements reflect reality of the public Tor network better when the measurement nodes are never connected directly, but always mediately over public Tor relays. In this case all controlled Tor nodes can run on the same local machine, configured with different ports and data directories.

The Tor processes can be configured to write detailed log statements to log files in their data directories. These can be used to track the most important events in making a hidden service available or accessing it. Times for single substeps can be measured by subtracting timestamps of log statements with a precision of up to 1 millisecond. In most cases these log statements should be sufficient to measure the substeps of the hidden service protocol. If intermediate steps are required or if log statements should contain more information, the Tor code can be extended; this was not necessary for the measurements performed in this thesis. The resulting data can then be analyzed with standard statistics software, like GNU R [79], as it was done in this thesis.

Tor nodes can be configured to either become part of the public Tor network or to create one's own private Tor network. The latter implies setting up one's own directory servers and overriding the default directory servers in the configurations of all participating Tor nodes. This process of setting up private Tor networks has been simplified and accelerated in the course of this thesis [38]. Setting up a private Tor network may be useful in order to test new features or to measure performance under laboratory conditions. However, the goal here was to obtain data and improve user experience in a realistic environment.

The task of configuring Tor processes and controlling their execution can be cumbersome. For some measurements it is necessary to create new Tor processes with new data directories and separate port configurations, so that they can run in parallel to other measurement runs. Therefore, the Java API PuppeTor[30] has been developed as part of this dissertation project. PuppeTor facilitates automatic configuration of Tor processes, including setting up private Tor networks. PuppeTor further contains a minimal HTTP server and HTTP client to automatically perform requests. The data directory of Tor processes controlled by PuppeTor are stored in a common working directory that is created for every execution of a program using PuppeTor. This organization facilitates the subsequent analysis of log files. It is not required to execute all Tor processes for a measurement setup with PuppeTor. In some cases it makes more sense to configure some Tor processes manually to run throughout the complete measurement time and start the remaining processes for every measurement using PuppeTor. The PuppeTor API has more recently been extended by Sebastian Hahn to a distributed test environment as part of the Google Summer of Code 2008 program.[31]

Figure 6.1 shows the basic types of the PuppeTor API. The `Network` is the central type that manages all controlled Tor nodes as well as client and server processes. Tor nodes are subdivided into the three types `ProxyNode`, `RouterNode`, and `DirectoryNode`. The standard configuration of a `ProxyNode` is to act as Tor client, that is, without relaying traffic for other users. The `RouterNode` contains a standard configuration

[30] The source code is available in the Tor repository at `https://svn.torproject.org/svn/puppetor/trunk` (last checked: Dec 17, 2008)

[31] See the accepted project application: `http://code.google.com/soc/2008/eff/appinfo.html?csaid=3A225E2DCCBA5B3A` (last checked: Dec 17, 2008)

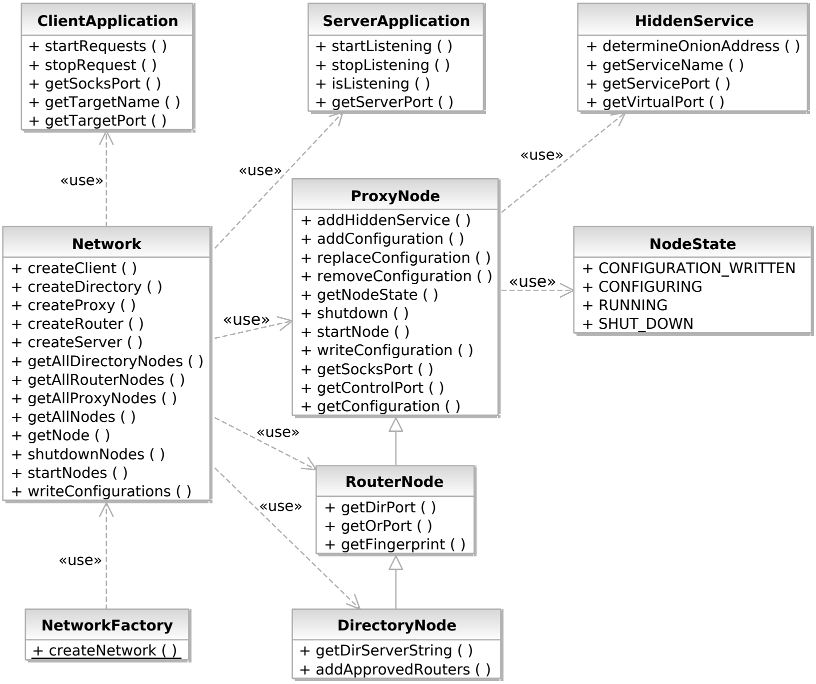


Figure 6.1: Basic types of the PuppeTor API

of a Tor relay. The DirectoryNode comprises the configuration template of a Tor directory server. The same executable file is used for all three roles which differ only in their configuration. A HiddenService includes all configurations that are necessary to offer a hidden service. The NodeState contains one out of four node states. The network may also contain client or server processes. The ClientApplication implements a minimal HTTP client that sends one or more HTTP requests to a given target and at a given interval. The ServerApplication contains a minimal HTTP server that answers HTTP requests.

Figure 6.2 shows an example application that uses PuppeTor to configure and control a single Tor process as part of the public Tor network. In

```
1. Network network = NetworkFactory.createNetwork("servpub", 7000);
2. ProxyNode proxy = network.createProxy("proxy");
3. proxy.addHiddenService("hidServ");
4. network.writeConfigurations();
5. network.startNodes(3 * 1000);
6. Thread.sleep(30 * 60 * 1000);
7. network.shutdownNodes();
```

Figure 6.2: Example application for measuring service publication times

this example, a Tor client is configured to offer a hidden service and run for 30 minutes. The code starts with creating a `Network` instance. The network is passed the initial port 7000 that is used to automatically assign consecutive port numbers to Tor nodes, hidden services, and server processes. In the next line a `ProxyNode` is instantiated with automatically assigned port numbers for its control port and its SOCKS port. In line 3 a hidden service is added to the node configuration with automatically assigned service port and virtual port. The complete configuration is written to disk in line 4. In line 5 the node is started with a timeout of 3 seconds for opening its control port. The executing PuppeTor thread then sleeps for 30 minutes while the Tor process publishes its service. Even though the hidden server could accept client requests, there is no server process in this example listening on the service port that is configured for the hidden service. Finally, the example is terminated after 30 minutes by shutting down the node in line 7.

6.2 Service Publication

The first measurement deals with hidden service publication. The service publication time is the time that is necessary to establish a hidden service in the Tor network and make it available for clients. This process includes four substeps: In the first step, the Tor client downloads directory information about available relays from the directory servers. After having downloaded at least 25% of all router descriptors, the Tor client establishes introduction points for its hidden service in the second step. In the third step, the Tor client waits for the established introduction points to become stable, that is, no new introduction points are established for a given time. Finally, in the fourth step, the Tor client opens circuits to the directories and uploads its hidden service descriptor. The total service publication time as defined here begins with starting the Tor process and ends with the first directory acknowledging receipt of a hidden service descriptor.

6.2.1 Measurements

Service publication times were measured by creating new Tor clients which offer one hidden service each. Hidden services were configured to use the distributed hidden service directory as described in this thesis. Tor clients were started with empty data directories, so that they do not have any current information about the network available before the measurements. Processes were started at an interval of 1 minute and stopped after 30 minutes. In this case, PuppeTor was used to configure the Tor processes to use separate data directories and to shut them down after 30 minutes. All processes were run on an Intel Core Duo 1.83 GHz laptop with 2 GB RAM running Linux and connected via DSL with 2 Mbit/s downstream and 192 Kbit/s upstream. Measurements were performed between July 10, 2008, 15:35 and July 11, 9:29 using Tor version 0.2.1.2-alpha-dev (r15806). Within this time, 1,068 data samples could be collected of which 2 runs failed and were excluded from later analysis.

The later analysis of measured data consists of extracting service publication times and the times for the four substeps from the Tor log files. Figure 6.3 shows an excerpt of one of the log files with the log statements that are relevant for the analysis. The service publication time can be cal-

```
1. Jul 10 15:40:01.705 [notice] Tor 0.2.1.2-alpha-dev (r15806) opening
   new log file.
2. Jul 10 15:40:06.659 [notice] We now have enough directory
   information to build circuits.
3. Jul 10 15:40:07.682 [info] rend_services_introduce(): Picked router
   dmass as an intro point for 2ml2xon5ch2l2j57.
4. Jul 10 15:40:08.692 [info] rend_services_introduce(): Giving up on
   dmass as intro point for 2ml2xon5ch2l2j57.
5. Jul 10 15:40:08.851 [info] rend_services_introduce(): Picked router
   b1qi954bcq34ob as an intro point for 2ml2xon5ch2l2j57.
6. Jul 10 15:40:13.548 [info] rend_service_intro_has_opened():
   Established circuit 11709 as introduction point for service
   2ml2xon5ch2l2j57
7. Jul 10 15:40:15.575 [info] rend_service_intro_established():
   Received INTRO_ESTABLISHED cell on circuit 11709 for service
   2ml2xon5ch2l2j57
8. Jul 10 15:40:46.028 [info] upload_service_descriptor(): Sending
   publish request for hidden service 2ml2xon5ch2l2j57
9. Jul 10 15:40:48.951 [info] connection_dir_client_reached_eof():
   Uploaded rendezvous descriptor (status 200 ("Service descriptor
   stored"))
```

Figure 6.3: Log statements of publishing a service in the network

culated by subtracting the timestamp of the first log statement from the timestamp of the last log statement; in the example the service publication time is 47.2 seconds which is rather quick as compared to the other runs.

Table 6.1 contains statistics of measured service publication times and its substeps which are explained below. Figure 6.4 visualizes all measured service publication times in a histogram. The measured mean time for service publication is 77.5 seconds, and 75% of all services were published within less than 80 seconds. However, there is also a small number of service publications that took between 200 and 800 seconds and which lead to high variability of service publication time. The highest service publication time in the measurements is 867 seconds.

Table 6.1: Measured service publication times (s)

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Download directory	1.936	4.636	7.148	26.440	31.810	798.000
Est. intro. points	2.220	9.728	13.000	15.620	18.840	75.170
Stabilize descriptor	13.290	30.750	31.040	31.000	31.320	31.990
Upload descriptor	0.359	1.767	3.192	4.429	5.320	100.300
Total	39.460	51.620	61.440	77.490	79.470	866.500

The separation of service publication time into its four substeps is meant to help understand the empirical distribution of measured times. The measured times for the substeps are also contained in Table 6.1. Figure 6.5 displays histograms of all four substeps. The four substeps are explained in the following.

The first step in publishing a hidden service is to *download enough directory information* to build circuits. The event that enough directory information have been downloaded is reported in the second log statement in Figure 6.3. The download includes the network status consensus and a fraction of at least 25% of all descriptors of running relays. The statistics for this substep show that downloading directory information is responsible for the high variability in service publication. While the mean time to download directory information is 26.4 seconds, the maximum time is 798 seconds. The sum of maximum values of all other substeps is slightly higher than 200 seconds, so that all higher values must result from high directory download times. The reason for these extreme values has turned out to be a very slow directory server: clients that downloaded the network status consensus from that directory server were delayed for a couple of minutes, while others succeeded within seconds. At the other end of values, the fastest directory download has been completed in 1.9 seconds.

In the next step the Tor client *establishes introduction points* for its hidden service. Therefore, the Tor client starts building circuits to three randomly

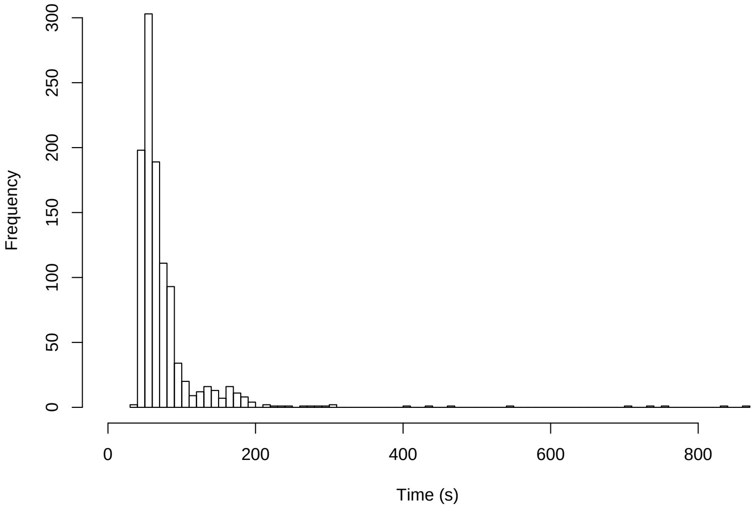


Figure 6.4: Measured service publication times

chosen relays as stated in log statement 3. If for some reason a circuit cannot be built or an introduction point not be established, the relay is given up as introduction point as shown in log statement 4. In this case, the Tor client picks another relay and starts establishing an introduction point on that relay instead. When a circuit is established as shown in log statement 6, the Tor client sends an EstablishIntro cell over the circuit requesting the relay to act as introduction point. The relay answers with an IntroEstablished cell which is indicated in log statement 7. The substep of establishing introduction points as measured here ends with the last successful establishment of an introduction point before making the first attempt to publish a descriptor.

In the measured data, establishing introduction points is performed in 15.6 seconds in the mean. The histogram shows that the frequency of introduction point establishment decreases starting with times around 15 seconds until approximately 50 seconds. Subsequently, there is a small

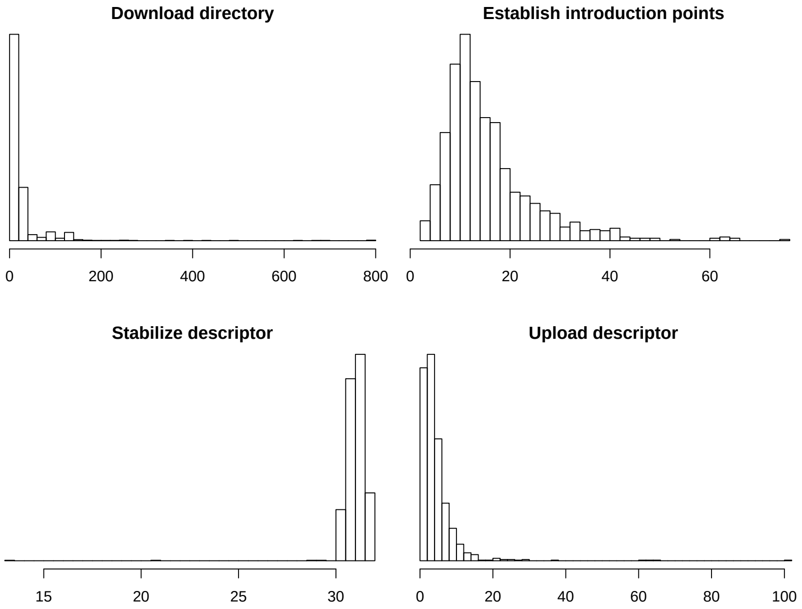


Figure 6.5: Components of service publication (x axes contain time (s), y axes frequency)

increase of cases between 60 and 66 seconds. The explanation for the increase at 60 seconds is a circuit creation timeout of 60 seconds: if circuit creation to one of the introduction points times out at 60 seconds and a subsequent attempt succeeds shortly afterwards, the latter event might conclude the introduction point establishment step; this further implies that previously established introduction points were not stable long enough to publish a descriptor. The maximum establishment time of introduction points is 75.2 seconds.

The third substep of service publication is the *stabilization time*. The intention is to wait for a certain time after the last introduction point establishment to publish a hidden service descriptor that is not likely to change shortly after publication. In the logs, the stabilization time is measured as the difference between the last introduction point establishment event in log statement 7 and the attempt to upload a descriptor in log statement 8. The current stabilization time is hard-coded to 30 seconds which can also be seen in the statistics. The minor deviation of one second to a mean value of 31 seconds results from implementation details, and a few values smaller than 30 seconds are attributable to a minor bug in the code.

The fourth and final substep is *uploading the hidden service descriptor to the directories*. The Tor client opens circuits to all directory nodes that are responsible for storing replicas of its descriptor. If pre-built circuits are available, the Tor client cannibalizes them and extends them by another hop. Otherwise, new 3-hop circuits are built to the directory nodes. After the circuit is open, the Tor client sends the hidden service descriptor which is acknowledged by the directory node. The descriptor upload time as measured here begins with the local decision to publish a descriptor in log statement 8 and ends with the first acknowledgment message received from a directory node in log statement 9.

The first descriptor upload is completed within only 4.4 seconds in the mean, and the vast majority of uploads succeeds within 30 seconds or less. However, there are single descriptor uploads that take over 60 or even up to 100 seconds. This value is surprisingly high, as multiple descriptor uploads are performed in parallel. The values over 60 seconds indicate a general problem in establishing circuits that are only overcome by timing out circuit establishments and making a second attempt. It turned out that there was a bug in the code resulting in failure of approximately 15% of all upload requests. In rare cases, all 6 replicas fell victim to this bug, so that neither of them could succeed at the first time.

6.2.2 Improvements

One of the goals of the measurements is to identify performance bottlenecks and to propose improvements. While service publication time may not be as important from a user perspective as connection establishment time, users expect their services to be available in a reasonable amount of time. It is important, however, to make only those changes that do not put excessive additional load on the network.

Downloading enough directory information to build circuits as the first substep of service publication exhibits a high mean time of 26.4 seconds and high variability with a maximum time of 798 seconds. Ensuring that directory servers have a minimum bandwidth is part of the solution to avoid extreme values in this substep. While it would make sense to further investigate and improve this substep, it is independent of the hidden service protocol and therefore out of scope at this point. The next substep, establishing introduction points, with a mean value of 15.6 seconds on the contrary is in scope. An acceleration of this step, especially avoiding extreme values of 60 seconds or more, is one option to speed up service publication. The third substep, stabilization time, is another candidate for lowering service publication time. It might be possible to reduce the fixed stabilization time of 30 seconds if variability of introduction point establishment can be reduced, too. The mean time of the last substep, uploading the descriptor, seems reasonable with only few options to improve. The extreme values of up to 100 seconds seem to result from a bug that has been fixed.

Build More Introduction Circuits than Needed

The high variability in establishing introduction points comes from the fact that the set of three introduction points is only considered stable when the third introduction point is established. That means that service publication time is determined by the slowest of the three introduction point establishments. If one of the circuit creations or cell transfers is delayed or fails and times out, service publication is delayed, too.

One approach to prevent single introduction point establishments from delaying the whole process is to parallelize this step. Instead of establishing only three circuits, the Tor client could build four, five, or six circuits to prospective introduction points and use only the first three opened introduction circuits to establish introduction points. The Tor client would only send EstablishIntro cells to the first three succeeding circuits and establish introduction points on them. The other relays to which circuits have been built would never learn that they were meant to act as introduction points. They would be kept for other purposes, like for being extended to rendezvous points and answering client requests later on. As a result, the additional network load is minimal, because these circuits would most likely have been built anyway.

The effect has been simulated based on the previously measured data. Therefore, 3,196 introduction circuit establishments have been extracted from the log files. These contain the times when an introduction point is picked, when the introduction circuit is opened, and when the IntroEstablished cell is received. If an introduction point is given up and a new relay must be picked as replacement, this still accounts for the first attempt to establish the introduction point. Otherwise, considered introduction point establishment times would be lower than they really are. Afterwards, simulations were performed with 10,000 runs each. In every run, n introduction point establishments were picked at random. From these, the three establishments with fastest introduction circuit opening times were selected. Only those introduction points were considered to be published in the first descriptor that were established before the stabilization time of 30 seconds expired. The last of these establishment times was then used as result of the simulation run. Simulations were performed for $n = 3 \ldots 6$.

The results of the simulations are shown in Table 6.2. The simulation of choosing 3 out of 3 introduction points has been performed to compare simulation results with the original data. The mean time of the simulation of 15.0 seconds is quite close to the measured value of 15.6 seconds.

Table 6.2: Simulated establishment times for the first 3 out of n introduction circuits

Introduction Points	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
3	0.466	9.040	12.460	14.990	18.500	73.630
4	0.549	8.023	10.550	12.340	14.730	70.880
5	1.410	6.876	8.822	9.924	11.420	67.230
6	0.602	6.053	7.810	8.430	9.915	43.500

The mean times of choosing 3 out of 4, 5, and 6 introduction points are 12.3, 9.9, and 8.4 seconds respectively. As a result, the overall service publication time would be accelerated by 2.7, 5.1, or 6.6 seconds in the mean, depending on n. Maximum establishment times would be reduced from 73.6 seconds to 70.1, 67.2, and 43.5 seconds respectively.

This change has been implemented and included in the Tor source code. A hidden server starts building 5 introduction circuits and picks only the first 3 that succeed for introduction point establishment. The reason for starting with 5 introduction circuits is that the Tor client would have built 2 internal circuits anyway to use them as rendezvous circuits later on. As a result, this change improves service publication time without affecting network load at all.

Change the Stabilization Time

The stabilization time of 30 seconds is another major substep that might be improved to reduce service publication time. Stabilization time as one of four substeps accounts for the largest share of service publication time with a mean time of 31 seconds. The choice of 30 seconds originally emerged from a guess of the developers. It has not been evaluated yet whether a shorter period might also suffice to upload a stable descriptor and make the service available more quickly. The other way round, it has not been investigated whether a longer delay might significantly reduce the number of necessary descriptor publications.

The measured data have been examined to derive theoretical service publication times for stabilization times between 1 and 90 seconds. Therefore, the logs of all previously measured runs were looked at. The introduction point establishment events (log statement 7 in Figure 6.3) were considered as events that restart the stabilization time. The first time when the stabilization time has elapsed was considered as the theoretical upload time of the first descriptor. This time excludes the upload time that is required to open a circuit and send the descriptor, which is assumed to be independent of the stabilization time.

Figure 6.6 shows theoretical mean times before uploading the first descriptor for stabilization times between 1 and 90 seconds. As reference point, the mean time before uploading the first descriptor for a stabilization time of 30 seconds is 71.8 seconds. Given the 77.5 seconds observed mean service publication time and subtracting mean descriptor upload time of 4.4 seconds and the implementation-specific deviation from 30 seconds stabilization time of 1 second, the difference between theoretical and observed service publication time is 0.3 seconds. The graph shows that mean times are almost linear for stabilization times in the interval from 10 to 50 seconds. Between 50 and 60 seconds there is a steep incline before times continue increasing linearly. Dividing stabilization time in half to 15 seconds would reduce mean time before the first descriptor upload by 17.8 seconds, while doubling stabilization time to 60 seconds would increase it by 43.6 seconds.

At the same time, reducing stabilization time leads to an increase in uploaded descriptors. Figure 6.7 visualizes the mean number of published descriptors within the first 30 minutes. For stabilization times between 10 and 50 seconds, the mean number of uploaded descriptors grows polynomially as a function of stabilization time. With a stabilization time of 30 seconds the Tor client needs to upload 1.4 descriptors in the mean,

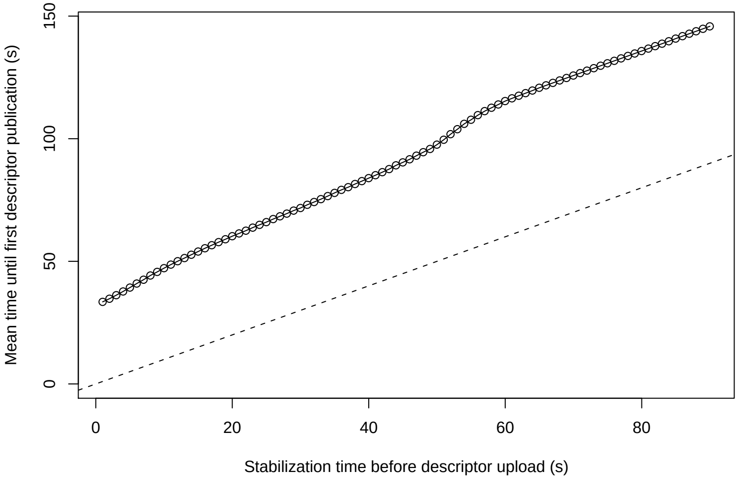


Figure 6.6: Theoretical mean times before first descriptor upload (circles) as a function of stabilization time (dashed line)

while this would change to 1.5 descriptors for a stabilization time of 15 seconds; an increase by approximately 10% (before rounding). Between 50 and 60 seconds, the mean number of uploaded descriptors drops to around 1.1 descriptors and stays almost constant for higher stabilization times. As a result, the number of descriptor publications could be reduced by roughly 20% (before rounding) when increasing stabilization time from 30 to 60 seconds.

As a result, there is a clear trade-off between the two objectives of publishing the first descriptor as quickly as possible and keeping the total number of descriptor publications low. While saving 17.8 seconds of service publication time when cutting down stabilization time to 15 seconds sounds promising, an incline of descriptor publications of 10% violates the stated policy of not increasing network load. This is why this change has not been included in the Tor source code.

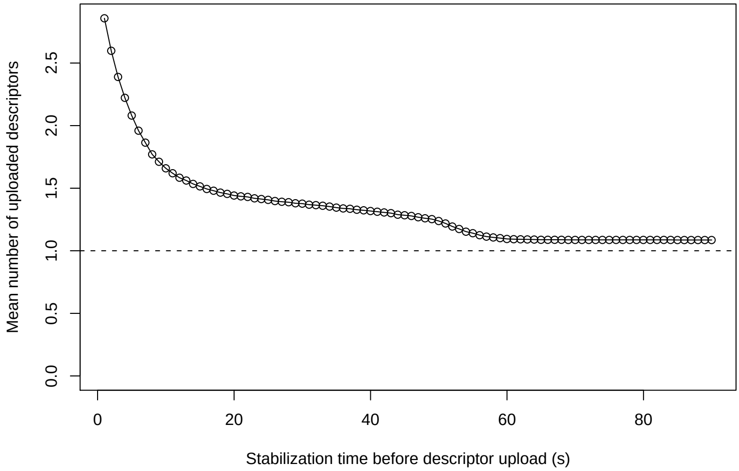


Figure 6.7: Theoretical number of uploaded descriptors (circles) as a function of stabilization time with fixed lower bound of 1.0 upload (dashed line)

6.3 Connection Establishment

The second part of measurements covers connection establishment, that is, the time that it takes for a client to establish a connection to a hidden server. Quick connection establishment times with low variability are an important requirement from a user perspective. Connection establishment begins when a user requests to open a connection to a hidden service and ends with establishment of a TCP connection between client and hidden server. Between these two events, multiple substeps need to be performed in order to establish a connection. The client first needs to open a directory connection and fetch a hidden service descriptor from one of the directories. After receiving the hidden service descriptor, the client establishes a rendezvous point and opens an introduction circuit in parallel. Next, the client sends an introduction request over the introduction circuit that is forwarded by the introduction point to the hidden server. The hidden server opens a circuit to the rendezvous point and sends a rendezvous message over it that is forwarded by the rendezvous point to the client. Finally, the client establishes an application-level stream over the rendezvous circuit that is acknowledged by the hidden server. Upon receiving this acknowledgement, the connection to the hidden server is established and the client can start sending application data.

6.3.1 Measurements

The measurement setup aims at measuring overall connection establishment times as well as 14 substeps. Therefore, all relevant points that are required for establishing a connection to a hidden server were controlled for the measurements to have access to their logfiles. These points include two Tor clients, one providing the hidden service and the other one accessing it, and two Tor relays, one acting as introduction point and one as rendezvous point. The source code has been modified to select the own introduction point and rendezvous point rather than picking relays at random. The only point that was not controlled in the setup is the directory server. While it would also have been possible to change the source code to force both client and hidden server to use a controlled node as directory, the expected additional results did not justify the required effort.

The three Tor processes for hidden service, introduction and rendezvous point were started prior to the measurements and kept running for the complete measurements. New clients were configured using PuppeTor with separate data directories and started at an interval of 25 minutes. Clients were allowed to download directory information and build circuits for 75 seconds before requesting them to establish a connection to the hidden server. Client, hidden server, and rendezvous point were run on a virtual root server with AMD Opteron 2.0 GHz processor and 400 MB RAM guaranteed running Linux. The introduction point was run on a distinct server with AMD Athlon 2.0 GHz processor with 1 GB RAM running Linux. All four points used Tor version 0.2.0.7-alpha.

Measurements were conducted between April 22, 2008, 16:00 and May 13, 22:20 by Christian Wilms [82] who generously made the log files available for further analysis in this thesis. Throughout this time, 1,197 connection establishment attempts were made of which 1,153 were successful (96.3%). 44 attempts failed for different reasons and were excluded from the statistical analysis. The own introduction point was used in 1,038 cases and the own rendezvous point in 1,070 cases.

Table 6.3 contains statistics of measured connection establishment times and its substeps. The mean time for establishing a connection to a hidden server is 33.8 seconds. While single attempts are performed in only 3.6 seconds at the minimum, the longest successful connection establishment took 191 seconds. Figure 6.8 shows a histogram of overall connection establishment times. One remarkable point in the histogram is the short but steep incline of values between 60 and 80 seconds. A distribution fit has shown that connection establishment times can best be described with a Frechét distribution, which is an extreme value distribution, for values less than 60 seconds and an exponential distribution for times greater than 60 seconds [41].

The total connection establishment times are useful to obtain an idea of the overall performance. But in order to understand the reasons for high delay and to suggest improvements, a more in-depth analysis of substeps is inevitable. Figure 6.9 visualizes relations of measured substeps for connection establishment. The numbers 10 to 29 constitute events in the log files on either client, rendezvous point, introduction point, or hidden server. The corresponding log messages of a successful connection establishment can be found in Figures 6.10 to 6.13. The labeled arrows denote measured substeps. Table 6.3 contains statistics of the measured substeps. Figure 6.14 shows histograms of the measured times. These substeps are discussed in more detail in the following.

Table 6.3: Measured connection establishment times (s)

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Open directory conn.	0.042	0.564	1.516	5.639	4.033	111.600
Fetch descriptor	0.061	0.558	1.431	2.521	2.738	106.900
Open rend. circuit (C)	0.000	0.000	0.001	2.322	0.001	85.880
EstablishRend.	0.010	0.110	0.267	0.918	0.776	56.100
Rend.Established	0.003	0.071	0.204	0.737	0.901	32.770
Open intro. circuit	0.035	0.625	1.763	7.058	4.300	100.900
Introduce1	0.010	0.340	0.744	1.100	1.144	42.170
IntroduceAck	0.017	0.218	0.765	1.404	1.471	24.150
Introduce2	0.013	0.236	0.570	1.401	1.154	37.260
Open rend. circuit (S)	0.095	1.741	3.132	5.703	6.310	48.530
Rendezvous1	0.021	0.552	1.106	2.455	2.116	37.150
Rendezvous2	0.003	0.092	0.228	0.712	0.798	26.230
Begin	0.193	1.316	1.980	3.411	3.108	79.040
Connected	0.074	0.918	1.587	3.019	2.965	57.320
Total	3.561	14.920	23.740	33.810	41.660	191.200

The first step in establishing a connection to a previously unknown hidden service is to *open a directory connection* to one of the three directory servers.[32] If a general-purpose circuit is already available, the client cannibalizes it and extends it by one hop to the directory server. Otherwise, the client builds a new 3-hop circuit ending at the directory server. After the circuit is open, the client sends a BeginDir cell to initialize a directory connection. The directory replies with a Connected cell. The mean time of these three steps is 5.6 seconds. This time is biased to a certain extent by a small number of extreme values. There is a fixed timeout of 120 seconds for opening the directory connection *and* fetching the descriptor in the next step. The timeout leads to a maximum observed value of 112 sec-

[32] The measurements have been performed with the server-based directory design, not with the distributed design proposed in this thesis.

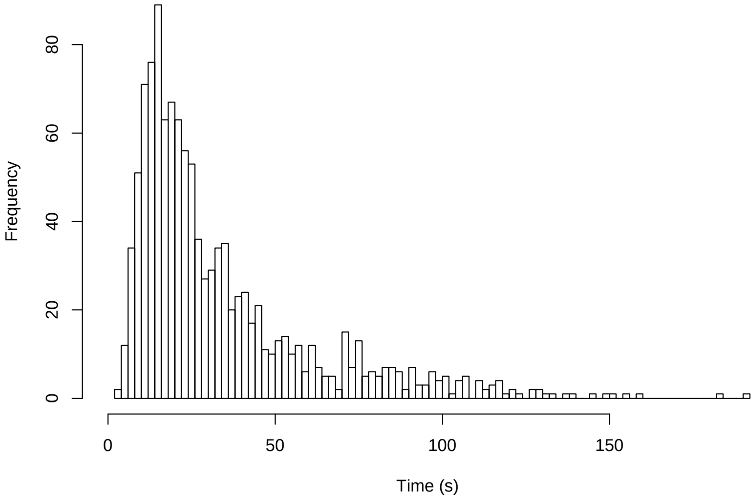


Figure 6.8: Total connection establishment time

onds for opening the directory connection. 16 of the 44 failed connection attempts were aborted due to the expiration of this timeout which makes this step the number one reason of failures.

In the next step, the client attempts to *fetch the hidden service descriptor* from the directory server. This step consists of sending a descriptor fetch message via the directory connection and waiting for the descriptor contained in the reply message. The measured time is the round-trip time of both messages, because the directory servers were not part of the measurement setup. This operation takes 2.5 seconds in the mean with only a few extreme values. The same 120-seconds timeout as for opening the directory connection leads to a maximum observed value of 107 seconds for fetching the descriptor. In 10 cases the connection establishment process failed, because the descriptor did not arrive before the timeout expired. 4 connection establishment attempts failed, because the descriptor could not be found on the directory server. The reason for this is most likely a

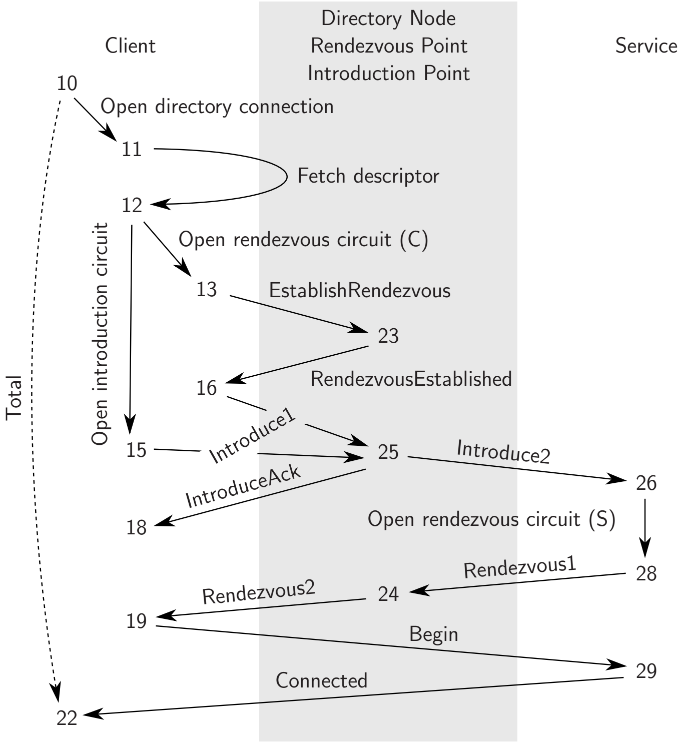


Figure 6.9: Measured steps in connection establishment process

restart of the requested directory server in the hour before the request.

After receiving a valid service descriptor and learning about at least one introduction point the client can take the next steps. These consist of establishing a rendezvous point and opening a circuit to one of the introduction points in parallel. The client starts with *opening a rendezvous circuit*. In the vast majority of cases the client cannibalizes an existing circuit and uses the last node in the circuit as rendezvous point. In these cases the operation returns immediately. The mean time for opening a rendezvous circuit is 2.3 seconds which is greatly influenced by the peak shortly after 60 seconds. 1,112 of 1,154 values are below 60 seconds with their max-

```
10. Apr 23 20:21:24.456 [info]
    connection_ap_handshake_rewrite_and_attach(): Got a hidden service
    request for ID ’xpw5lcjsag7u6l6w’
11. Apr 23 20:21:36.393 [info]
    connection_edge_process_relay_cell_not_open(): ’connected’ received
    after 1 seconds.
12. Apr 23 20:21:37.905 [info] connection_dir_client_reached_eof():
    Received rendezvous descriptor (size 402, status 200 ("OK"))
13. Apr 23 20:21:37.906 [info] rend_client_rendcirc_has_opened():
    rendcirc is open
14. Apr 23 20:21:37.906 [info] rend_client_send_establish_rendezvous():
    Sending an ESTABLISH_RENDEZVOUS cell
15. Apr 23 20:21:38.240 [info] rend_client_introcirc_has_opened():
    introcirc is open
16. Apr 23 20:21:38.811 [info] rend_client_rendezvous_acked(): Got
    rendezvous ack. This circuit is now ready for rendezvous.
17. Apr 23 20:21:39.078 [info] rend_client_send_introduction(): Sending
    an INTRODUCE1 cell
18. Apr 23 20:21:39.469 [info] rend_client_introduction_acked():
    Received ack. Telling rend circ...
19. Apr 23 20:21:46.791 [info] rend_client_receive_rendezvous(): Got
    RENDEZVOUS2 cell from hidden service.
20. Apr 23 20:21:47.097 [info] connection_ap_handshake_attach_circuit():
    rend joined circ 1353 already here. attaching. (stream 10 sec old)
21. Apr 23 20:21:47.098 [info] connection_ap_handshake_send_begin():
    Address/port sent, ap socket 18, n_circ_id 1353
22. Apr 23 20:21:52.801 [info]
    connection_edge_process_relay_cell_not_open(): ’connected’ received
    after 5 seconds.
```

Figure 6.10: Log statements of connection establishment as observed on a client

```
23. Apr 23 20:21:38.494 [info] Established rendezvous point on circuit
    3307 for cookie 4815A117
24. Apr 23 20:21:45.680 [info] Completing rendezvous: circuit 26085
    joins circuit 3307 (cookie 4815A117)
```

Figure 6.11: Log statements of connection establishment as observed on a rendezvous point

```
25. Apr 23 20:21:39.305 [info] Received an INTRODUCE1 request on circuit
    49661
```

Figure 6.12: Log statements of connection establishment as observed on an introduction point

```
26. Apr 23 20:21:39.857 [info] rend_service_introduce(): Received
    INTRODUCE2 cell for service "xpw5lcjsag7u6l6w" on circ 64317.
27. Apr 23 20:21:39.873 [info] rend_service_introduce(): Accepted intro;
    launching circuit to "$094C0337F5A03A9D62B35358DDD9F3A8E48FF23B"
    (cookie 4815A117) for service xpw5lcjsag7u6l6w.
28. Apr 23 20:21:44.042 [info] rend_service_rendezvous_has_opened():
    Done building circuit 64529 to rendezvous with cookie 4815A117 for
    service xpw5lcjsag7u6l6w
29. Apr 23 20:21:49.458 [info] connection_exit_begin_conn(): begin is
    for rendezvous. configuring stream.
```

Figure 6.13: Log statements of connection establishment as observed on a hidden server

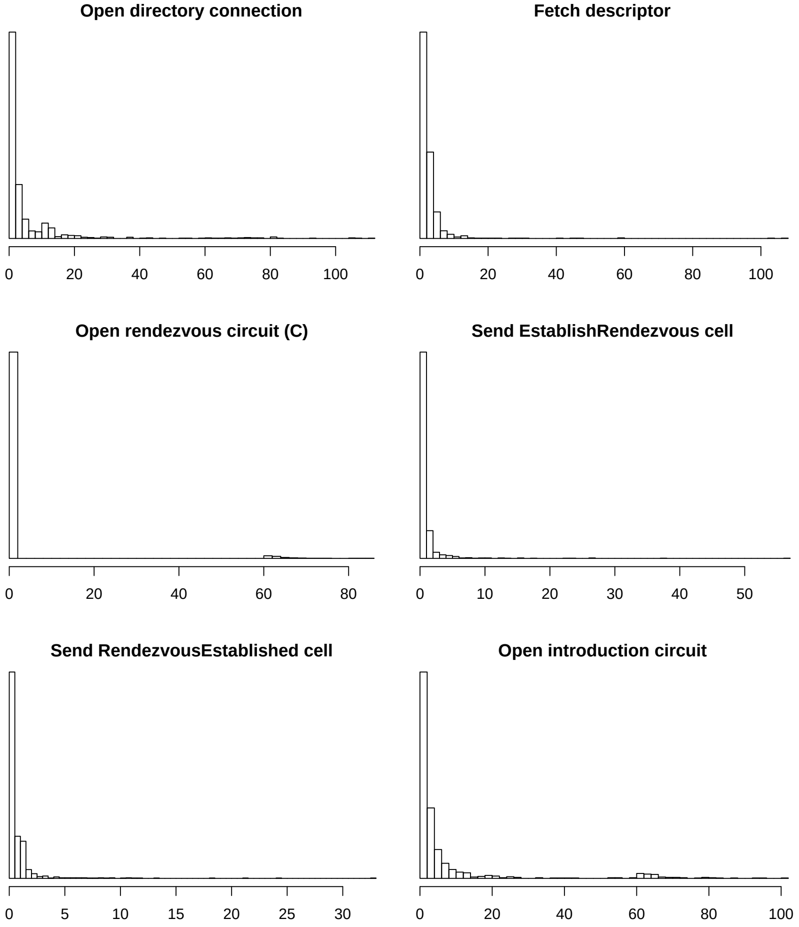


Figure 6.14: Components of connection establishment (x axes contain time (s), y axes frequency)

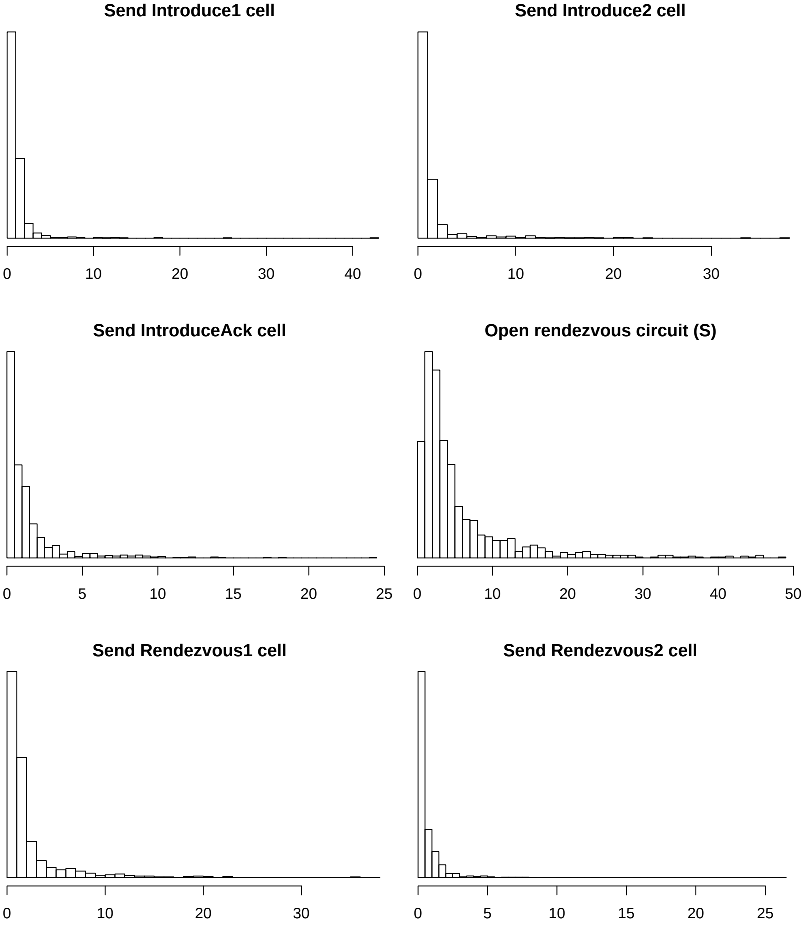


Figure 6.14 – continued from previous page

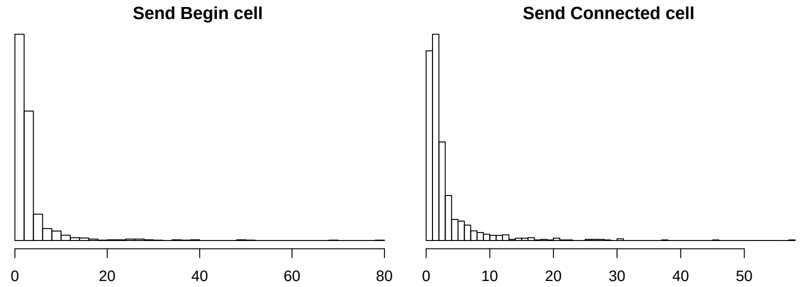


Figure 6.14 – continued from previous page

imum value at 1.5 seconds. The reason is a timeout of 60 seconds that starts after receiving the hidden service descriptor. If the connection to the hidden server has not been established within this time, a second attempt is made which includes establishing a new rendezvous point. Hence, the 42 extreme values can be explained as being the times between starting the first attempt and succeeding in the second attempt.

In the next step the designated rendezvous point needs to learn about its acting as rendezvous point and therefore has to be told the rendezvous cookie. The client *sends an* EstablishRendezvous *cell* over the rendezvous circuit which is then answered by the rendezvous point with a RendezvousEstablished cell. The mean transfer times of these cells are 0.9 and 0.7 seconds, respectively. There are only very few extreme values in both transfer times. In cases when the client made a second connection attempt, only the second transfer time of these cells has been included in the statistics, so that there are no values greater than 60 seconds.

In parallel to establishing the rendezvous point, the client tries to *open a circuit to one of the introduction points*. While pre-built circuits can be cannibalized for this purpose, too, they need to be extended by an additional hop to the introduction point. If no circuit is available for cannibalization, a new 3-hop circuit is built ending at the introduction point. Opening an introduction circuit takes 7.1 seconds in the mean which is again biased by the values greater than 60 seconds. The 60-seconds timeout leads to giving up circuit establishment to the introduction point and starting all over again. This situation occurred in 63 of 1,153 connection attempts.

After the rendezvous point is established and the introduction circuit is open, the client *sends an* INTRODUCE1 *cell* along the introduction circuit containing the address of the rendezvous point and the rendezvous cookie. Upon receipt the introduction point *forwards the* INTRODUCE2 *part* of it to the service and *replies with an* INTRODUCEACK *cell* to the client. The mean values for these steps are 1.1, 1.4, and 1.4 seconds, respectively. As with transfer times of ESTABLISHRENDEZVOUS and RENDEZVOUSESTABLISHED cells, there are only very few extreme values.

The hidden server, upon receiving an INTRODUCE2 cell, *opens a rendezvous circuit* to the rendezvous point specified in the INTRODUCE2 cell. In most cases the service can cannibalize an existing circuit and extend it by one hop. Otherwise, the service builds a new 3-hop circuit ending at the rendezvous point. The mean time to open the rendezvous circuit is 5.7 seconds. The distribution of values is comparable to opening the introduction circuit on client side for values below 60 seconds. The difference between these two steps is that failing rendezvous circuits are not retried by the hidden service. In case of a failure the client retries the request.

After the rendezvous circuit is open, the hidden server *sends a* RENDEZVOUS1 *cell* over the circuit to the rendezvous point. It contains the rendezvous cookie which is validated by the rendezvous point. If it matches with the cookie of a previously received ESTABLISHRENDEZVOUS cell, the rendezvous point strips off the rendezvous cookie and *forwards the cell as* RENDEZVOUS2 *cell* to the client. The mean transfer times of RENDEZVOUS1 and RENDEZVOUS2 cells are 2.5 and 0.7 seconds.

When receiving the RENDEZVOUS2 cell, the client knows that the rendezvous circuit has been extended to the hidden server. However, this does not complete connection establishment yet. The client still needs to *send a* BEGIN *cell* over the rendezvous circuit to open a TCP stream for the transported application protocol. Upon receiving a BEGIN cell, the service opens a connection to the actual service and *responds to the client with a* CONNECTED *cell*. The receipt of the latter finally concludes the connection establishment process. Sending the two cells takes 3.4 and 3.0 seconds, respectively. Later analysis has revealed that sending the BEGIN cell is artificially delayed by 0.5 seconds on average due to a bug, so that the means of both transfer times in a fixed version would be roughly the same.

The steps after sending the INTRODUCE1 cell and before receiving the RENDEZVOUS2 cell happen invisibly for the client. The client relies on a request timeout of 60 seconds that starts with receiving the descriptor to detect failures. If this timeout expires before a RENDEZVOUS2 cell is received, the request is given up and a second attempt is made. In total, a client makes two requests before assuming that a hidden service is unavailable. The remaining 14 failed connection attempts all were delayed or failed at either of these steps. In a few cases the RENDEZVOUS1 arrived at the rendezvous point which, however, could not forward it, because the client had already closed the circuit.

6.3.2 Improvements

The analysis of substeps has revealed that the primary bottlenecks of connection establishment are those steps in which circuits need to be opened to previously unknown nodes. These critical steps include opening a directory connection and an introduction circuit on client side as well as a rendezvous circuit on server side. These steps take 5.6, 7.1, and 5.7 seconds in the mean, respectively. In most cases these steps consist of cannibalizing a previously built 3-hop circuit and extending it by a fourth hop to a directory node, introduction point, or rendezvous point. The extend step exhibits huge variance and failure rate which leads to necessary retries. All three steps are on the critical path of connection establishment, so that a way to reduce these bottlenecks would most likely lead to an improvement of connection establishment times. Possible solutions could be eliminating circuit opening steps, reducing timeouts to detect failures earlier, or starting multiple attempts in parallel.

The second category of substeps that account for large delays are sending cells over circuits with many hops. Both BEGIN and CONNECTED cells are transported over a 6-hop circuit between client and server. The mean times for these steps are 3.4 and 3.0 seconds. While it seems logical that a cell transfer takes more time the longer the circuit is, it might be useful to think about combining cells to eliminate cell transfers on the critical path of connection establishment.

Again, it is important that improvements do not increase the general load on the network excessively. Further, changes need to retain the same security properties as the original protocol.

Reduce Timeout and Parallelize Directory Connection and Descriptor Fetch

The clients in the performed measurements request the hidden service descriptor from the centralized hidden service directory. They make a single attempt by sending their request to one of the three directory servers at random. If the request fails or times out, it is not repeated at the other directory servers. The result is that 26 out of 44 failures in the measurements do not succeed within the timeout of 120 seconds. In 4 cases the hidden service descriptor is not found on the requested directory server.

The decentralized hidden service directory as proposed in this thesis changes this to a certain extent. Clients retry their fetch requests at all directory nodes that are responsible for storing a replica of the hidden service descriptor. These retries are required to overcome node failures and single corrupt directory nodes. While retries do not prevent single requests from timing out, they reduce failure cases due to not finding a descriptor.

The next step could be to introduce a timeout for descriptor fetches, so that failures are detected more quickly than in the current 120 seconds. Another approach would be fetching descriptors in parallel from two or more directory nodes. Parallel fetches could also be delayed by a certain time to give the first fetch request the chance to succeed before generating additional network load by making a second attempt. An advantage of the decentralized directory is the fact that the additional requests would be distributed to all directory nodes, not only to three servers.

Alas, the effects of timeouts and parallelization cannot be quantified with the measured data. The three hidden service directory servers have different characteristics as compared to the nodes in the distributed directory. It is expected that a higher percentage of directory requests fails in the distributed directory which is then compensated by retried requests. While it can be assumed that timeouts and parallelization reduce descriptor fetch times, an evaluation is up to future work.

Reduce Timeout and Parallelize Opening Client-Side Introduction Circuit

The same improvements as for directory connections can also be discussed for the opening time of client-side introduction circuits. As opposed to directory fetches, clients utilize a timeout of 60 seconds for opening an introduction point before making a second attempt. However, the probability of an attempt to succeed after 30 seconds decreases significantly. Figure 6.15 visualizes the fraction of successfully extended introduction circuits for given times. In this graph, only the 93.8% of attempts that succeed in the first try are shown; attempts that succeed in the second try or fail are excluded. Merely 12 of 1,093 introduction circuits have been opened after 30 seconds, which is a fraction of only 1.1%.

The effect of reduced timeouts for opening introduction circuits on overall connection establishment has been simulated using the previously measured data. Table 6.4 shows simulated introduction circuit opening

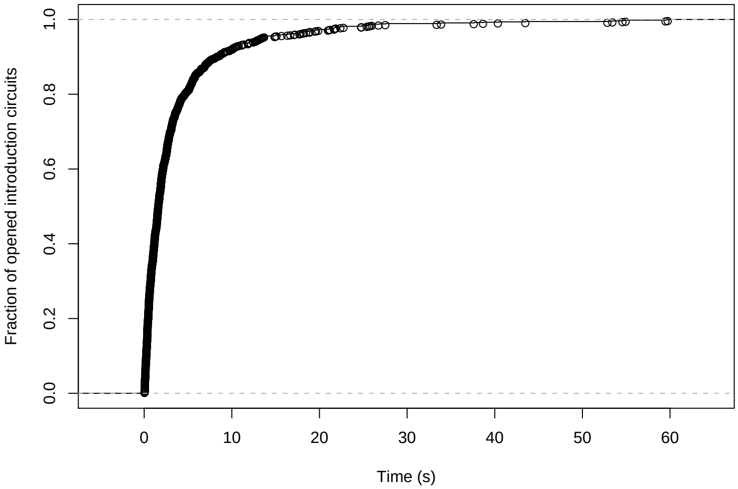


Figure 6.15: Empirical cumulative distribution function of opening client-side introduction circuits

times for various timeouts between 5 and 60 seconds. The overall timeout remains 120 seconds as in the original protocol, but with shorter timeouts more than 2 attempts can be made, and retries can be started earlier. The mean time of using a timeout of 60 seconds is 8.5 seconds and by that 1.5 seconds higher than the originally measured mean time of 7.1 seconds. One possible explanation is that the simulation also includes values close to 120 seconds which affect the mean value. Such values have been excluded from the original measurements with only successful connection establishment attempts; if opening the introduction circuit took most of the 120 seconds, the subsequent steps were not likely to succeed within the remaining time. The mean times for timeouts of 45, 30, or 15 seconds are 7.9, 6.5, and 4.9 seconds. Further, the maximum times are reduced from 120 seconds to 116, 112, and 88 seconds. This reduction also increases the probability that connection establishment succeeds in later steps before the 120-seconds timeout runs out.

Table 6.4: Simulated introduction circuit opening times (s) for reduced timeouts (s)

Timeout	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
5	0.035	0.699	1.951	3.393	5.222	42.210
10	0.035	0.697	1.968	4.182	5.486	70.900
15	0.035	0.705	1.962	4.885	5.532	88.410
20	0.035	0.713	1.979	5.459	5.486	102.000
25	0.035	0.689	1.951	5.938	5.479	117.900
30	0.035	0.699	1.968	6.465	5.499	112.400
45	0.035	0.713	1.987	7.895	5.499	115.800
60	0.035	0.692	1.942	8.520	5.379	119.800

At the same time, smaller timeouts might lead to discarding circuits that would have been completed with a higher timeout. Table 6.5 shows the number of attempts to open introduction circuits for different timeouts. The mean number of circuits that are necessary to contact an introduction point are 1.077 for the current timeout of 60 seconds and 1.095, 1.107, and 1.152 for timeouts of 45, 30, and 15 seconds. The result is an increase of circuit establishments of 1.7%, 2.8%, or 7.0% for these three reduced timeouts.

Another approach to accelerate opening of introduction circuits is to open a second circuit in parallel to a different introduction point. Such a parallel request can be done after a certain delay, so that the network load is not doubled or multiplied. If the first circuit does not succeed within a given time, it is not given up, but another attempt is started in parallel. Whichever circuit extension succeeds first is used for introduction, while the other circuits are torn down. The difference to simply reducing the timeout is that circuits might still succeed when other attempts are made in parallel.

Table 6.6 shows simulated introduction circuit opening times for paral-

Table 6.5: Simulated introduction circuit opening attempts for reduced timeouts (s)

Timeout	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
5	1.000	1.000	1.000	1.372	2.000	9.000
10	1.000	1.000	1.000	1.198	1.000	8.000
15	1.000	1.000	1.000	1.152	1.000	6.000
20	1.000	1.000	1.000	1.128	1.000	6.000
25	1.000	1.000	1.000	1.112	1.000	5.000
30	1.000	1.000	1.000	1.107	1.000	4.000
45	1.000	1.000	1.000	1.095	1.000	3.000
60	1.000	1.000	1.000	1.077	1.000	2.000

lel attempts with delays from 5 to 60 seconds. The mean values for delays of 45, 30, and 15 seconds are 7.8, 6.5, and 4.8 seconds. These values are slightly lower than the times for reduced timeouts, which are 7.9, 6.5, and 4.9 seconds, but not significantly. The maximum times for a delay of 15 seconds is 76.6, and by that 11.8 seconds lower than with a reduced timeout of 15 seconds.

Table 6.7 shows the number of attempts that are made in the simulated cases with different delays. Mean values are 1.095, 1.105, and 1.146 for delays of 45, 30, and 15 seconds. Again, these mean values are only slightly lower than those for reduced timeouts of 45, 30, and 15 seconds.

The two approaches to reduce timeouts and to perform further attempts in parallel after a certain delay can be combined with the delay being lower than the timeout. The result would be that a certain number of parallel attempts would be performed in quick succession while limiting the number of parallel attempts at a time. Possible parameters could be a delay of 15 seconds for parallel attempts and a timeout of 30 seconds.

Lenhard [35] has performed measurements of connection establishment to hidden services in low-bandwidth environments. In those mea-

Table 6.6: Simulated introduction circuit opening times (s) for delayed parallel circuit establishment (s)

Delay	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
5	0.035	0.713	1.991	3.110	5.174	30.380
10	0.035	0.699	1.968	4.036	5.479	50.110
15	0.035	0.692	1.968	4.780	5.491	76.640
20	0.035	0.713	1.979	5.376	5.491	82.160
25	0.035	0.683	1.962	5.922	5.486	116.400
30	0.035	0.713	1.979	6.467	5.512	114.800
45	0.035	0.713	1.968	7.839	5.486	115.400
60	0.035	0.705	1.979	8.606	5.412	119.800

Table 6.7: Simulated introduction circuit opening attempts for delayed parallel circuit establishment (s)

Delay	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
5	1.000	1.000	1.000	1.325	2.000	7.000
10	1.000	1.000	1.000	1.185	1.000	6.000
15	1.000	1.000	1.000	1.146	1.000	6.000
20	1.000	1.000	1.000	1.124	1.000	5.000
25	1.000	1.000	1.000	1.112	1.000	5.000
30	1.000	1.000	1.000	1.105	1.000	4.000
45	1.000	1.000	1.000	1.095	1.000	3.000
60	1.000	1.000	1.000	1.078	1.000	2.000

surements clients were connected via cell phone or telephone networks when connecting to a hidden service. The evaluation has revealed that the 90th percentile of introduction circuit opening times is 38.2 and 35.3 seconds for cell phone and telephone network, respectively, compared to 20.4 seconds for a broadband connection. Lenhard suggests to use a timeout of 45 for opening the client-side introduction circuit. This approach seems to be reasonable given that users might want to access hidden services via both broadband and low-bandwidth connections.

The changes have been implemented and included in the Tor source code. At the time of writing, a client uses a timeout of 30 seconds instead of 60 for opening a circuit to an introduction point and starts a second attempt in parallel after a delay of 15 seconds. However, following the findings by Lenhard, the 30-seconds timeout should be increased to 45 seconds. The idea is to accelerate connection establishment for clients on fast connections while still allowing clients on slow connections to complete requests within the given timeout.

The proposed optimizations apply only to opening client-side circuits and should not be adopted to the server-side rendezvous circuit. A hidden service might be requested by many clients at the same time. The server should therefore not spend more resources than necessary, but rely on clients to perform retries in time. If a hidden server would use multiple circuits per client, this would also make denial-of-service attacks on hidden servers more attractive for an adversary.

Combine Introduction and Rendezvous Circuits

The most radical design change to improve connection establishment goes back to an idea from Øverlier and Syverson [58]. They proposed two modified hidden service protocols that reduce the number of involved relays in connection establishment. Their ideas rely on the concept of *valet nodes* [57] that constitute an additional protection of introduction points, but which are not essential for their protocol modifications. The protocol changes as described here are adapted to the original hidden service protocol without the valet node concept.

The first protocol change combines a large part of the client-side introduction and rendezvous circuits to a single circuit. The rendezvous point is established on the third hop on a circuit before it is extended by a fourth hop to the introduction point. Upon receiving an introduction request, the hidden server opens a rendezvous circuit to the third hop of the client-side circuit. The result is a 6-hop circuit between client and service that is used exclusively for one client-service connection. However, this protocol still requires three circuit extensions as the original protocol does. Therefore, it is unlikely that the protocol improves connection establishment times. The only effect is that the client consumes 2 circuits instead of 3 which does not affect connection establishment times, as new circuits can be created in the background. This first protocol change is therefore not considered in the following discussion.

The second protocol change goes a step further and combines the roles of introduction point and rendezvous point. The hidden servers continue to establish introduction points, and clients keep opening circuits to them in order to establish a connection. But in contrast to the original protocol there is no distinct rendezvous point. The same 6-hop circuit that is used for introduction is also used for the rendezvous cells and transporting the actual application streams. This change saves the circuit extension step to the rendezvous point and is therefore a good candidate to reduce the delay in connection establishment. A possible design that combines the roles of introduction and rendezvous point in the original hidden service protocol has been specified in the course of this thesis [42]. A pre-evaluation of this design change performed by Wilms [82] has revealed that it reduces connection establishment times by 8.3 seconds or 24.4% in the mean.

The downsides of the second protocol change are multiple security issues that need to be solved. One of the original reasons for the separation of introduction and rendezvous points was that a relay shall not be made responsible for relaying data on behalf of a certain hidden service [11]. Responsibility changes when the same introduction point learns about the hidden service and is later used to relay (encrypted) application data. Two changes to the protocol can ensure that an introduction point can deny knowledge of the hidden service identity that it works for: The first change is using a fresh introduction key for each introduction point as proposed in the previous chapter. As a result the introduction point does not learn the hidden service identity automatically. But the introduction point could still access the hidden service and find out from the content what kind of data it transports. Therefore, as the second change, the hidden server could include an introduction cookie in the hidden service descriptor and require clients to include it in the encrypted INTRODUCE2 cells. This way an introduction point cannot access the hidden service anymore to find out what contents are served. Hence, the introduction point cannot be made responsible for storing possibly unwanted contents. However, there are still open questions to legal liability that require more discussion.

Another possible security problem is that an adversary who can attack an introduction point, for example by performing a distributed denial-of-service attack [15], does not only eliminate one access point to the hidden service. The attack would also terminate all current client connections to the hidden server using that introduction point.

The combination of introduction and rendezvous circuit further reduces the number of unknown relays on service side for the connection between client and service from 3 to 2. With a separate rendezvous point that is chosen by the client, the service cannibalizes an existing 3-hop circuit and extends it to the rendezvous point. In the changed design the combined introduction and rendezvous point as the third hop of a circuit built by the service is known to the client, and thereby to a possible adversary. This change facilitates traffic analysis attacks where an adversary attempts to trace back a circuit to its origin to locate a hidden server. On the other hand, an extension of service-side introduction circuits from 3 to 4 might reduce the performance improvement that was gained with this design change.

Combine Introduction with Opening Application-Level Stream

The last proposed design change addresses the BEGIN and CONNECTED cells that need to be sent over the 6-hop circuit between client and hidden server. These two cells are required to attach an application-level stream to the circuit, so that application data can be sent and received. However, it is the main purpose of building a circuit between client and hidden server to attach at least one application-level stream to it. The idea is to drop necessity of sending these two cells and integrate their contents in the INTRODUCE2 and RENDEZVOUS2 cells. The hidden server could cache the BEGIN cell that is part of the INTRODUCE2 cell and use it upon receiving the first data packet from the client. The client would conclude from the CONNECTED cell that is included in the RENDEZVOUS2 cell that the stream is open and start sending application data to the hidden server. This design change would save $3.4 - 0.5 + 3.0 = 5.9$ seconds in the mean (transmission time of the BEGIN cell, corrected by the bug that caused a delay of 0.5 seconds, plus transmission time of the CONNECTED cell). An evaluation of this design change is up to future work.

6.4 Conclusion

In this chapter a setup has been proposed to measure performance of operations in the Tor network with special focus on hidden services. The basic approach is to run a few Tor nodes as part of the public Tor network, perform the desired operations, and evaluate the resulting log files. In this regard, the Java API PuppeTor has been presented as a tool for automatically configuring and executing Tor processes.

Two operations of Tor hidden services have been analyzed in detail, namely making a hidden service available in the network and establishing a connection to it. The measurements have shown that service publication takes 77.5 seconds in the mean. The main reason is a fixed stabilization time of 30 seconds that has the purpose of reducing the number of descriptor uploads whenever the set of introduction points changes. The next cause is the time for downloading directory information which takes 26.4 seconds on average. Establishing introduction points takes 15.6 seconds in the mean. Improvements have been suggested to accelerate the establishment of introduction points and the stabilization time. The changes to introduction point establishment have been implemented and included in the Tor source code.

Further, the process of establishing a connection to a hidden server has been analyzed in more detail. The overall process takes 33.8 seconds in the mean. The main bottleneck in this process is opening circuits to previously unknown relays. In total there are three substeps which are affected by this problem, two of them on client side and one on hidden server side. Improvements have been proposed for the two client-side steps of opening circuits to directory nodes and introduction points. Further, two major changes to the hidden service protocol have been described, one of them combining introduction and rendezvous point and the other one combining the introduction process with opening the first application stream. The improvements to opening circuits to introduction points on client side have been implemented and added to the Tor source code.

The major causes of bad performance of Tor hidden services have been identified and some of them enhanced. Some performance bottlenecks are inherited from the nature of multi-hop routing in anonymous communication networks. In these cases hidden services would benefit from improvements to the circuit-building process. Although the focus of these measurements is on Tor hidden services, it can be expected that similar designs would have similar performance problems and should therefore take the results of this analysis into account.

7 Related Work

Up to this point, the description comprised background work and the contribution of this thesis. The work presented in the background chapters was selected in order to provide the necessary details to understand the contribution chapters. In addition to the background work there is more work that is related to the contribution but which is not necessarily required for understanding. Some of this work has been mentioned briefly in the contribution chapters. This related work shall be described in this chapter to complete the picture of the state of the art of research on pseudonymous services, especially Tor hidden services. Where applicable, the presented related work is compared with the contribution of this thesis.

7.1 Private Hidden Services

Øverlier and Syverson [57] propose an extension of Tor hidden service descriptors to support private hidden services which is probably the closest related work to this thesis. By coincidence, their work has been developed almost at the same time as a similar approach by the author of this thesis [39]. The latter has been published only two months prior to the work of Øverlier and Syverson at a non-related conference. Both developments are therefore considered as independent. The fact that two similar approaches on providing private hidden services have been developed roughly at the same time underlines the importance of the problem.

Øverlier and Syverson propose the extension of Tor hidden service descriptors to so-called *contact information tickets*. These tickets have two important security properties: Only those clients which have been told the onion address of a hidden service can locate the tickets *and* understand the contained list of introduction points. Furthermore, if required, the hidden server can issue tickets for specific clients only and exchange a secret cookie with them to locate the ticket and understand its content. If required, ticket identifiers can change periodically in order to hide when the service was offered for the first time. These properties are achieved by using the service descriptor index $hash(address +' 1' + cookie + date)$ for storing and locating tickets and $hash(address +' 2' + cookie + date)$ for encrypting its content. Without knowing the input to these hash functions, one cannot generate current or future identifiers or encryption keys. In order to verify updated tickets at the directory, the authors propose to use a reverse hash chain scheme. The initial publication of a ticket is accompanied by an iterated hash value $v_n = hash^n(v)$ with $v = address +' 1' + cookie + date$. Subsequent updates $k = n - 1 \dots 1$ contain the predecessor in the hash chain of the last publication $v_k = hash(v_{k-1})$ with $v_1 = v$. This scheme ensures that only the hidden server is able to update its ticket at the directory servers. The authors briefly discuss to store tickets in a distributed hash table instead of the directory servers.

The idea to encrypt contact information tickets bears an important problem that is mentioned but not solved by the authors: Although the directory servers can detect false updates using the reverse hash chain scheme, they cannot detect false initial publications. An adversary that learns about a valid lookup identifier could easily store a false descriptor with own protection against false updates. Even though clients would not accept this descriptor, the hidden service would be unavailable for them. Such a situation can occur when a hidden service is unavailable for a certain time and the lease for a previously stored descriptor expires. In that case a directory server cannot verify future descriptors anymore and has to believe that the false descriptor is legitimately stored under the given identifier. (The other way round, if directory servers would store descriptors or iterated hash values for an unlimited time, another adversary could mount an attack on the directory servers by publishing large numbers of useless descriptors.) This situation could also occur when an adversary pre-occupies the slots for identifiers with date components in the future. The approach that was taken in this thesis is to leave some parts of the descriptor content unencrypted, so that directory servers can verify that descriptors have been created by the holder of the claimed hidden service identity. The drawback, however, is that this makes descriptors linkable to a certain extent. A solution that provides confidentiality *and* authentication is left to future work.

7.2 Locating Hidden Servers

The primary security property that hidden services provide is to hide the location of the hidden servers. An adversary shall not be able to link the identity of a hidden service to the location of the server that provides the hidden service. Consequently, most proposed attacks on hidden services aim at locating hidden servers. These attacks include a variation of the predecessor attack [83] and exploit the fact that Tor hidden servers build circuits to rendezvous points for all connection requests. These attacks can be accelerated by manipulating the path selection process of Tor. Further, Murdoch [51] presents a way to locate hidden servers by their clock skew. These attacks and related approaches for countermeasures are discussed in the following.

Predecessor Attack. The predecessor attack has first been described for the Crowds system by Reiter and Rubin [68]. Later, Syverson and others [77] examined a related attack that worked in Onion Routing systems. Wright and others described the predecessor attack for various anonymous communication systems in [83] and have further formalized it in [85]. Even though Tor hidden services had not been deployed at this time, the attack on the original Onion Routing design [25] has similar properties as a predecessor attack on Tor.

The predecessor attack exploits the fact that the initiator of a connection selects a new path through the network in regular intervals. An adversary that controls a limited number of nodes in the network has a certain probability that one or more of her nodes are selected in some of these paths. The goal of the adversary is to be selected at the first and last position of a path in order to link the initiator with the responder of a connection. The adversary records the predecessors of the first controlled node in a path. Whichever predecessor is observed most often is most likely the initiator of the connection. For multiple observations, this probability can only increase over time.

Wright and others propose a defense for the predecessor attack by removing randomness from part of the node selection process [84]. They introduce so-called *helper nodes* that are fixed at certain positions for all selected paths. There can be fixed helper nodes for the first position, for the last position, or for both first and last positions. The rationale is that if either or even both positions are occupied by helper nodes, the adversary cannot succeed with the predecessor attack. This defense is limited by the fact that an initiator could select a node controlled by the adversary as helper node. This would defeat the protection and even facilitate the attack. However, the probability of an initiator picking a bad node as helper node is fixed while the probability of a successful predecessor attack increases over time.

Øverlier and Syverson [56] applied the predecessor attack to Tor hidden services. This attack does not even require to control two relays in a circuit, but only one. The authors exploit the fact that a hidden service needs to build new circuits to the rendezvous points that are selected by clients that wish to contact the hidden service. For every circuit the hidden server selects three relays at random. The goal of the adversary is to be picked as first relay in such a circuit. The authors further make use of the fact that Tor selects relays for circuit creation based on their self-advertised bandwidth capacity and uptime. They changed the source code to have their relay advertise far more bandwidth than it really has in order to improve the probability of being selected by the hidden server. There is no need to control the last relay in this attack, because the adversary also runs the client which is able to detect traffic patterns in the connection to the hidden service. Traffic analysis is facilitated by the fact that the authors reduced the circuit length to the rendezvous point from 3 to 1 by changing the source code. As a result, there are four relays between hidden service and client of which the rendezvous point is selected by the adversary and the relay next to the client is controlled by the adversary, too, if the attack succeeds. In a variation of the attack the authors further control the rendezvous point. This variation allows the adversary to determine the position of her controlled relay in the path more quickly. As a countermeasure to their attack, the authors propose to adopt the concept of helper nodes, which are called *entry guards* in Tor. Entry guards are selected once during the first startup of a Tor process and are fixed for all later path selections.

This attack shows a major problem with Tor hidden services: A hidden server is forced to create new circuits to rendezvous points in order to answer client requests. Although the hidden server can select the relays for these circuits itself, this can be exploited by an adversary. Entry guards are an effective countermeasure against this threat. In addition to that, hidden services that perform client authorization as described in this thesis are protected against this attack as well.

Manipulation of Path Selection Process. Another problem which facilitates attacks on the location of hidden services is the fact that an adversary can advertise relays with false bandwidth capacities and uptimes. By doing so, the attacker influences the path selection process of victims, making it more likely that they pick relays which are controlled by the adversary.

When creating a new circuit, the Tor clients randomly select a series of relays from the network status. The way how relays are selected is referred to as the *path selection algorithm.* In an early deployment state of the Tor network, relays were selected with uniform probability in order to achieve maximum anonymity. With the increase of relays run by volunteers this algorithm had to be changed, because the relays had very different bandwidth and uptime properties. A uniform selection leads to performance bottlenecks, as the relays with small resources have been overloaded soon while the resources of others were not fully used. As a solution, the path selection algorithm was changed so that clients weight relays by their available bandwidth that is listed in the network status.

Øverlier and Syverson [56] have first exploited the fact that available bandwidth is self-advertised by the relays. They changed the source code, so that their relay advertised a much higher bandwidth and uptime than it really had in order to be selected more often by clients. Bauer and others [2] also make use of this vulnerability in order to perform the predecessor attack in the Tor network for non-hidden-service connections. They further reduce the necessary resources of an adversary by performing traffic analysis only based on circuit establishment messages. In cases when circuit establishment messages reveal that an attack cannot be successful, they abort the establishment process and force the victim to build a new circuit, possibly one that can be attacked successfully. They find from practical evaluations in a private Tor network that in a network of 60 honest relays and 6 relays controlled by an adversary, 46% of all circuits could be compromised. However, this excludes the use of entry guards. Borisov and others [4] quantify the loss of anonymity caused by compromised relays that selectively disrupt circuits which an adversary cannot compromise.

Bauer and others [2] propose a few countermeasures against adversaries lying about their resources. Directory servers could check uptime by periodically sending a heartbeat message to relays and put a cap on the maximum uptime that they believe in. Further, directory servers should verify bandwidth claims by periodically sending probes and actively measure the bandwidth. The authors further suggest that the maximum number of relays that may be run on a single IP address should be limited to prevent Sybil attacks [13]. Snader and Borisov [74] propose improvements to measuring available bandwidth of Tor relays and to select paths accordingly. They state that bandwidth should be measured by each relay keeping track of the peak bandwidth that it has seen for directly connected relays. In contrast to active measurements, this does not put an extra load on the network for probing.

Murdoch and Watson [53] simulate the consequences of different path selection algorithms on security and performance of anonymous communication systems like Tor. They discuss an adversary who has access to a botnet and therefore controls a large number of IP addresses with each of them having only small bandwidth. They come to the conclusion that the bandwidth-weighted path selection algorithm in Tor exhibits both best performance and best security properties against a botnet-based attack. The authors' conclusion makes clear how difficult it is to design a path selection algorithm that gives sufficient protection against different kinds of adversaries.

Clock-Skew Attack on Tor Hidden Services. Murdoch [51, 52] proposed another attack to reveal the location of a hidden server that is unrelated to the predecessor attack. This attack relies on the fact that the skew of computer clocks changes with varying CPU temperatures. The author describes an attack in which an adversary influences CPU temperature of a hidden server by putting high load on them for a certain period of time. High load forces the server to perform many cryptographic operations which in turn increase CPU temperature. In periods when no requests are sent, the CPU cools down again. The result is a recognizable pattern in clock skew that can be observed by an adversary. Murdoch assumes that the adversary can establish direct TCP connections to candidate servers who might provide the hidden service. The adversary then can compare TCP timestamps to see whether these servers are under high load or not. As a result, the adversary can confirm which of these servers hosts the hidden service with a certain probability. Zander and Murdoch [86] have improved the technique to reveal hidden services using the same attack principle. The result is that clock-skew attacks can be performed faster with fewer network traffic. An effective protection against this attack is to impede direct connections to a hidden server. This protection can be achieved by simple firewall or network address translation. Fortunately, hidden services do not require the hidden server to accept incoming connections from clients, but only to establish outgoing connections to relays.

7.3 Attacks on Availability of Hidden Services

In addition to locating attacks, hidden services also need to resist attacks with the attempt to make them unavailable for legitimate clients. The Tor design paper [11] states that hidden services are protected against distributed denial-of-service attacks and that attackers are forced to attack the whole network because they do not know the IP address of the hidden server. However, multiple attacks on the availability of hidden services have been discussed in the literature or observed in the deployed Tor network.

Attacks on Introduction Points. The first points of attack on a hidden service are its introduction points. Øverlier and Syverson [57] propose an extension of the Tor hidden service protocol to better protect introduction points from distributed denial-of-service attacks. So-called *valet nodes* are the central concept of their proposal: A valet node lies between the client and an introduction point and has the purpose of hiding the location of the introduction point from the requesting client. The client establishes a connection to a valet node and sends to it the encrypted location of an introduction point. The valet node decrypts this location and extends the circuit to the introduction point which in turn forwards messages to the hidden service. Therefore, the hidden server generates a fresh *service key*, sends the private key to the introduction point and the public key to the clients within hidden service descriptors. This enables a client to validate authenticity of a node being an introduction point for a hidden service, but without revealing the identity of that node.

Although the valet node approach constitutes an interesting extension of Tor hidden services, it is questionable whether introduction points are subject of attacks. If an adversary manages to shut down an introduction point, the hidden server notices this immediately and can establish a new introduction point on another relay. The hidden server then advertises this change by uploading a new hidden service descriptor containing the updated list of introduction points. Further reasons against integration of valet nodes to the protocol changes described here are the resulting complexity of both design and implementation and additional delay to connection establishment.

Attacks on Hidden Servers. Distributed denial-of-service attacks on the hidden services are a more serious threat. An adversary can exploit the fact that a hidden server needs to create new circuits to rendezvous points to answer requests. The adversary would send fake connection requests to the hidden service, so that the hidden server cannot answer legitimate requests anymore. The reason is that building circuits requires to perform expensive public-key cryptography which limits the number of circuits that a Tor client can build at a time. Further, the hidden server is unable to distinguish between legitimate and false requests. The necessary resources of the adversary can be reduced even further by reducing circuit length to an introduction point from 3 to 1. In October 2008, the first denial-of-service attacks on hidden services could be observed and were brought to attention in the official Tor IRC chatroom. There is no evidence that the attackers performed their attack as described here, but it is quite likely that they did so. There are no defenses against this attack yet, besides requiring client authorization as described in this thesis which, however, is not applicable for all hidden services. A possible mitigation might be to radically change the hidden service protocol by removing the step where the hidden server builds a circuit to a rendezvous point [58]; but this would introduce a couple of new problems as discussed in Section 6.3.2.

Attacks on Tor Relays. Fraser and others [15] discuss distributed denial-of-service attacks on Tor relays. They exploit the fact that starting to establish a TLS connection is an expensive operation for a Tor relay, but not for the initiator of the connection. An adversary who can send a large number of TLS connection requests to a relay might be able to prevent legitimate participants from establishing a connection to that relay, thus making it unavailable. The authors propose a mitigation of such attacks by using client puzzles. These make TLS connection establishments more expensive for clients and the attack too expensive and therefore unattractive for an adversary. Distributed denial-of-service attacks on relays are not directly related to hidden services. But an adversary could attack entry guards or introduction points that are used by a hidden service and force it to pick new ones, possibly controlled by the adversary. Such an attack on introduction points is prevented by the client authorization protocols proposed in this thesis, because an adversary—besides an adversary being an authorized client—is unable to locate the introduction points of a service.

7.4 Applications Based on Hidden Services

The main application for hidden services is assumed to be web services. This assumption cannot be proven, though, because there is no way to investigate the applications provided using hidden services. The only way to do so would be to illegitimately collect onion addresses and attempt to access them, which has not been done here. A similar study has been performed by McCoy and others [48] to observe usage of Tor by possibly illegitimately sniffing user traffic on Tor exit nodes and observing clients connecting to entry nodes. For hidden services such an approach raises even more ethical questions.

A few applications other than web services have been publicly announced which are built on top of hidden services. Two of these applications are described here which are related to this thesis by illustrating possible applications of private services.

Torchat. The *torchat* project[33] that was started in November 2007 is a server-less instant messaging system on top of Tor hidden services. All users in the torchat system set up an own hidden service and are uniquely identifiable by the onion address of their service. Users exchange onion addresses and contact other users by establishing connections to their hidden services. As a result, the system hides locations of all users, keeps their communication confidential, and hides relations between users. The project does not, however, take special precautions to hide presence of its users once they have disseminated their onion address. Anyone who has learned about a user's onion address can track that user's presence without her authorization or even knowledge. The extensions to perform client authorization for hidden services would benefit the security properties of torchat significantly.

OnionCat. The OnionCat project[34] uses Tor hidden services to build an IP-based virtual private network. OnionCat was first announced on the

[33] See the project homepage: `http://code.google.com/p/torchat/` (last checked: Dec 17, 2008)

[34] See the project homepage: `http://www.abenteuerland.at/onioncat/` (last checked: Dec 17, 2008)

Tor mailing list in June 2008. Participants set up a hidden service which constitutes their identity in the virtual private network. They further set up an OnionCat instance which acts as a transparent proxy between their computer and the hidden services of other participants. OnionCat maps 80-bit onion addresses to 128-bit IPv6 addresses. OnionCat permits the execution of arbitrary IP-based protocols rather than TCP-based protocols. An OnionCat participant contacts another OnionCat user by opening a connection to that user's hidden service which is identified by a given IPv6 address. All further IP communication between the two participants is then tunneled through the hidden service connection.

The OnionCat approach is to create an anonymous global network which permits connections between all participants. However, this raises concern with regard to security: An adversary could easily connect to any OnionCat user and attempt to attack the services behind it. Further, users leak their presence when running OnionCat and have no possibility to restrict their hidden service and the IP-based services behind it to certain users only. Again, utilization of hidden services with client authorization is a possible solution to these security problems.

This chapter has presented work that is related to the contribution of this thesis. Related work includes approaches to make Tor hidden services more private, various attacks on either revealing the location of hidden services or making them unavailable, and proposed applications based on hidden services. The next chapter concludes this thesis.

8 Conclusion

The motivation for this thesis was the insight that Internet services which are provided by private persons require specific protections which are less important for services provided by enterprises or organizations. The first use case for private services that came to mind were server-less instant messaging systems. They permit users to exchange presence information and text messages directly rather than using an allegedly trusted server. Server-less instant messaging systems return control over presence status information to the users and protect them from leaking this information to untrusted entities. In addition to that, there are all kinds of services that private persons might want to provide, including web servers, IRC servers, file servers, and so on. Whenever a service is linked to a private person, all information about the service can be used to reveal information about its provider as well.

The approach to this problem was to use privacy-enhancing technologies to achieve better protection for private services. Privacy-enhancing technologies, especially anonymous communication networks, help protect the link between content of communication and communicating entities. The approach was to provide a private service as pseudonymous service in an anonymous communication system. Such an approach prevents anyone from finding the real location of the computer that provides the service, thus not allowing untrusted entities to correlate the service to the person who provides it. This property is a necessary prerequisite in the attempt to hide privacy-relevant attributes such as service activity or frequency of clients accessing the service. Further, it prevents an adversary from mounting an attack on the private service in the attempt to shut it down.

An analysis of privacy-enhancing technologies with the ability to provide pseudonymous services has revealed that none of the existing designs meets all requirements of private services. The first requirement is low-latency message transfer that allows for interactive protocols. The design further needs to support long-term responder pseudonyms, that is, users may offer a service under a persistent name which cannot be linked to the location of the server. These requirements narrow down the selection of existing designs to the TAZ/Rewebber system [23], the Pseudonymous IP network [19], the Invisible Internet Project (I2P), and Tor hidden services [11]. From these four designs, the latter three would be candidates for interactive services, while the TAZ/Rewebber system was designed for static content, like web pages. None of the remaining three candidates supports private services by design, as services might leak activity to unauthorized clients or become victim to distributed denial-of-service attacks. There is no way to configure a service so that it can exclusively be used by authorized clients. From the three systems, Tor hidden services have turned out to be the best candidate for an extension towards private services. Tor is actively used by hundreds of thousands users, has an active community, and the Tor developers are open to discuss changes and accept patches if proven to be useful.

Contribution. This thesis has presented three contributions to improve Tor that are necessary to implement private services with hidden services as primary building block: distributed storage for hidden service descriptors, client authorization as part of the hidden service protocol, and performance improvements of service publication and connection establishment.

The first contribution of this thesis is a *distributed storage for hidden service descriptors.* The centralized storage based on three directory servers has turned out to lack some important requirements that are necessary to support private services. First and foremost, the centralized design does not permit private entries which can only be located and understood by authorized clients. A distributed storage permits to store entries on changing nodes which are only known to the server and authorized clients. Distribution helps conceal hidden service activity and usage which would become apparent from observing requests to central directory servers. Encryption of at least part of the entries provides for an effective defense against unauthorized access attempts on private services. A distributed approach is further more resistant against censoring particular services which could be performed easily in a centralized setting. Finally, a distributed design has the potential to scale to larger numbers of stored entries and requests which is a property that a centralized approach does not have.

The proposed distributed storage design bears some resemblance to distributed hash tables, in particular to Chord [76]. Storage nodes are assigned unique identifiers in the same identifier space as hidden service descriptors. The architecture of Tor permits to make use of a router list to determine responsibility and routing of requests. As a result, directory nodes are not required to exchange routing information among themselves. They further do not share stored entries in order to mitigate the risk of disseminating private entries to untrusted directory nodes. Relying on a central routing table that is only updated every three hours requires the storage nodes to be notably stable. It further requires a certain amount of replication to compensate node failures and untrustworthy nodes. Possible security problems have been discussed including some measures to counter them. An evaluation of the Tor node population over a time of almost three months has shown, among other things, that the longer a Tor relay is connected to the network, the smaller is the probability that it leaves within the next three hours. Especially relays with a minimum uptime of 24 hours exhibit a very low churn rate and are therefore suitable for being used in a distributed storage. The evaluation further helped deduce a useful replication rate that is necessary to guarantee a reasonable overall availability with high probability. The proposed design has been implemented and deployed in the public Tor network in November 2007. One year later, on November 30, 2008, 84 of 1,191 Tor relays are running as hidden service directory nodes. This makes it presumably one of the largest deployed directory for long-term responder pseudonyms at the time of writing.[35]

The second contribution of this thesis aims at *performing client authorization* as part of the hidden service protocol which is required for private services for several reasons: Non-authorized clients need to be excluded from accessing the service. While this would also have been achieved by performing client authorization *after* connection establishment, such an approach has disadvantages: Non-authorized clients would be able to make access attempts to a service before the actual authorization takes place. Non-authorized access attempts make the service vulnerable to all kinds of attacks, including locating [56] and distributed denial-of-service attacks. Another problem that arises from performing client authorization subsequently after connection establishment is that information about service activity and usage is disseminated to various untrusted points in the network. These include introduction points, directory nodes, and former clients. An approach that conceals service activity and usage may reveal as few information about the identity of a hidden service. An extension of the hidden service protocol is necessary to perform client authorization as part of the connection establishment process.

This extension has been subdivided in two parts: The first protocol extension focuses on hiding the hidden service identity from the introduction points, encrypts the list of introduction points in hidden service descriptors, and requires clients to authorize themselves when establish-

[35] Possible exceptions to this are peer-to-peer-based systems, like I2P, where all users participate in storing descriptors.

ing a connection to the service. This extension is a compromise between improved security properties and efficiency in the sense of requiring no more messages than the original protocol. The second protocol extension uses client-specific hidden service identities, makes use of private entries as supported by the distributed storage, encrypts the list of introduction points in hidden service descriptors, and delays descriptor publication so that links between client identities are hidden to a certain extent. This protocol exhibits the best security properties at the price of limited scalability. A security analysis has evaluated both protocol extensions compared to approaches using the unchanged hidden service protocol and subsequent client authorization. The necessary modifications for both protocol changes have been implemented and included in the development releases of Tor. This pseudonymous services design is the first to support client authorization. While client authorization may not be desired for the majority of services, there may be future applications that can make good use of them. Private services constitute only one example. Moreover, it might be useful to perform client authorization even for public services as a means to gain better protection against denial-of-service attacks.

The third contribution of this thesis focuses on *performance improvements* of hidden service publication and connection establishment. Setting up and accessing a hidden service performs considerably worse than doing so with a directly accessible Internet service. One reason is that relaying traffic over multiple hops adds a large amount of delay to connections between client and hidden server. Furthermore, complexity of the hidden service protocol adds more delay, as numerous steps need to be accomplished to establish a connection to a hidden server. It can be assumed that bad performance of hidden services deters a lot of people from using them, because they are not willing to wait. Bad performance reduces the anonymity set, thus decreasing the anonymity that hidden service users get. An improvement of hidden service performance is therefore not just convenience but a means to enhance protection of both service providers and clients.

Part of the work on improving hidden service performance has been the establishment of a measurement setup. A Java-based API, PuppeTor, has been created that can be used to configure and control Tor processes for measurements. This API has been made freely available in 2007. Others have started to extend this API to a distributed testing environment for Tor processes. Measurements have revealed that establishing three introduction points that are required for a hidden service is one of the main reasons for delay in service publication. An improvement has been suggested and implemented that increases the number of concurrently established introduction points and uses only the first three that succeed. Further, measurements of the connection establishment process have shown that circuit building steps are the main bottleneck in the hidden service protocol. Improvements have been proposed and implemented which reduce the circuit building timeout for client-side introduction points and makes parallel circuit building attempts after a certain delay. The main insight of working on hidden service performance is that circuit building has a major influence on setting up and accessing a hidden service. Even extending circuits by a single hop can delay connection establishment significantly. These are the first measurements of Tor hidden service performance so far. Now that the major bottlenecks are known, more work can be done on mitigating them. Possible examples are combining substeps in the hidden service protocol or accelerating circuit building performance in general. Besides of improving hidden service performance, the insights gained here could help in designing future pseudonymous service systems as well.

All improvements to Tor hidden services presented here have been inspired by the idea to implement private services like the instant messaging example as stated at the beginning of the thesis. Based on these modifications it should be straightforward to build such a system using hidden services that perform client authorization. In addition, other private services, for example, private web or file services can be realized using the improved hidden services. More important, the work on Tor hidden services is intended to advance research on pseudonymous services in general. There have been a few designs for low-latency responder pseudonymity in the past of which Tor hidden services are the most advanced concept to date. One can assume that there will be further enhancements to Tor hidden services as well as newly developed systems in the future.

A Implementation

The following tables contain all patches that have been included in the Tor codebase in the context of this dissertation project. Size denotes the number of lines in a patch, including added and removed lines as well as context before and after changes. The contents of a revision N can be downloaded from the Tor repository with the following command:

```
svn diff https://svn.torproject.org/svn/tor/trunk/ -r <N-1>:<N>
```

Table A.1: Features added to implement the distributed descriptor storage

Date	Description	Revision	Size
May 1, 2007	Enable directory authorities to record hidden service usage.	r10067 r10068 r10084	812 206 30
Aug 11, 2007	Improve hidden-service-related log statements.	r11074 r11077	138 13
Sep 18, 2007	Prepare cryptography functions for v2 rendezvous descriptor format.	r11489 r11490	393 43
Sep 19, 2007	Drop unused v1 rendezvous descriptor format.	r11496 r11498	492 227
Sep 21, 2007	Improve cryptography functions and add tests.	r11538	433
Oct 18, 2007	Update specification in rend-spec.txt to reflect new v2 rendezvous descriptor format.	r12007 r12027	217 165
Oct 28, 2007	Support encoding and parsing of v2 rendezvous descriptor format.	r12254 r12255	1113 717
Oct 29, 2007	Allow directory mirrors to act as v2 hidden service directories and make hidden services use them.	r12272	1353
Oct 31, 2007	Generate v0 and v2 descriptors in parallel.	r12299 r12300	741 139

Table A.1 – continued from previous page

Date	Description	Revision	Size
Nov 1, 2007	Make clients fetch descriptors from v2 hidden service directories.	r12302	254
		r12303	56
Nov 8, 2007	Perform cleanups as preparation for 0.2.0.10-alpha release.	r12319	358
		r12361	620
		r12388	215
		r12431	28
Dec 21, 2007	Perform various cleanups, refactorings, and documentations as preparation for 0.2.0.13-alpha release.	r12527	45
		r12528	39
		r12603	53
		r12604	22
		r12607	802
		r12608	157
		r12609	26
		r12715	1100
		r12825	437
		r12826	99
		r12841	32
		r12842	21
		r12900	1004
		r12901	42
Jan 16, 2008	Perform cleanups as preparation for 0.2.0.16-alpha release.	r13147	351
Jan 24, 2008	Make clients re-fetch v2 rendezvous descriptors if they failed before.	r13250	296
		r13251	99
		r13253	30
Jan 25, 2008	Do not fail, if there are too few v2 hidden service directories.	r13263	85
Jan 25, 2008	Do not re-fetch v2 rendezvous descriptors, if a v0 rendezvous descriptor is already there.	r13265	152
Jan 26, 2008	Perform cleanups and refactorings as preparation for 0.2.0.18-alpha release.	r13269	134
		r13270	14
		r13271	35
		r13287	383
		r13293	15

Table A.1 – continued from previous page

Date	Description	Revision	Size
Sep 12, 2008	Change default value for acting as v2 hidden service directory to true.	r16858	76
Sep 24, 2008	Drop requirement of being a directory mirror in order to act as v2 hidden service directory. Make all relays v2 hidden service directories by default.	r16961	69

Table A.2: Bugfixes added while implementing the distributed directory storage

Date	Description	Revision	Size
Sep 19, 2007	Clients should specify their chosen rendezvous point by identity digest rather than by (potentially ambiguous) nickname.	r11499	46
Sep 19, 2007	Hidden services were choosing introduction points uniquely by identity digest, but when constructing the hidden service descriptor they merely wrote the (potentially ambiguous) nickname.	r11500 r11501	25 22
Nov 14, 2007	When lacking a consensus, do not try to perform rendezvous operations.	r12493	69
Nov 17, 2007	When authorities detected more than two relays running on the same IP address, they were clearing all the status flags but forgetting to clear the HSDir flag. So, clients were being told that a given relay was the right choice for a v2 hidden service directory lookup, yet they never had its descriptor because it was marked as 'not running' in the consensus.	r12509 r12515	26 48
Nov 17, 2007	The HSDir flag in v3 network status consensus documents was cleared when there are too many relays at a single IP address. Clear it in v2 network status documents too, and also clear it when the relay is no longer listed in the relevant network status document.	r12522	103

Table A.2 – continued from previous page

Date	Description	Revision	Size
Dec 1, 2007	The v2 hidden service descriptor format allows descriptors that have no introduction points. But Tor crashed when trying to build a descriptor with no introduction points (and it would have crashed when trying to parse one).	r12579 r12580 r12619	137 86 26
Dec 22, 2007	Fix a crash when accessing hidden services: it works for the first time a given introduction point is used, but on subsequent requests garbage memory would be used.	r12913	27
Jan 6, 2008	Complain less at both the client and the relay when trying to upload a hidden service descriptor to a relay that used to have the HSDir flag but does not have it anymore.	r13037	106
Feb 13, 2008	Re-fetch v2 (as well as v0) hidden service descriptors when all introduction points for a hidden service have failed.	r13431 r13439 r13492	69 21 66
Feb 13, 2008	Make the v2 hidden service code respect the SafeLogging setting.	r13493	279
Feb 17, 2008	Resolve problems with (re-)fetching hidden service descriptors.	r13540	116
Apr 15, 2008	Avoid a rare assert that can trigger when Tor does not have much directory information yet and it tries to fetch a v2 hidden service descriptor.	r14373	28
Sep 24, 2008	If clients or hidden services do not use BEGIN_DIR cells, they should not attempt to contact v2 hidden service directories with non-open directory port.	r16960	42

Table A.3: Features added to implement client authorization

Date	Description	Revision	Size
Aug 9, 2008	Allow hidden services to configure authorization data for clients.	r16475 r16477 r16479	857 13 266

Table A.3 – continued from previous page

Date	Description	Revision	Size
Aug 12, 2008	Allow clients to configure authorization data for hidden services.	r16510	258
Sep 1, 2008	Hidden services publish descriptors for their authorized clients.	r16598 r16599 r16604 r16706	772 24 26 23
Sep 24, 2008	Clients fetch descriptors for hidden services that perform client authorization.	r16955	1755

Table A.4: Bugfixes added while improving hidden service performance

Date	Description	Revision	Size
Jun 11, 2008	In very rare situations new hidden service descriptors were published earlier than the minimum of 30 seconds after the last change to the introduction points.	r15113 r15114	16 14
Jun 12, 2008	While setting up a hidden service, some valid introduction circuits were overlooked and abandoned.	r15149 r15151	26 14
Jun 17, 2008	When establishing a hidden service, introduction points that originate from cannibalized circuits were completely ignored and not included in rendezvous service descriptors.	r15332 r15335	51 13
Jul 6, 2008	Attach connections immediately upon receiving a RENDEZVOUS2 or RENDEZVOUSESTABLISHED cell rather than waiting for the next execution of a 1-second loop.	r15699	46
Jul 11, 2008	When a hidden service is giving up on an introduction point candidate that was not included in the last published rendezvous descriptor, do not reschedule publication of the next descriptor.	r15825	67

Table A.4 – continued from previous page

Date	Description	Revision	Size
Aug 5, 2008	Mark configuration options RendNodes, RendExcludeNodes, HiddenServiceNodes, and HiddenServiceExcludeNodes as obsolete, because they never worked properly.	r16144 r16401	271 59
Aug 5, 2008	In some edge cases, the router descriptor of a previously picked introduction point becomes obsolete. Give up on it rather than continually complaining that it has become obsolete.	r16404	49
Sep 16, 2008	When fetching hidden service descriptors for different versions in parallel, do not fail the whole request when one version fails.	r16915 r16916	122 99
Sep 23, 2008	Almost 1/6 of all requests to the hidden service directory nodes failed, because the required router descriptor has not been downloaded yet. Hold back the request until the router descriptor has been downloaded.	r16808 r16810 r16817 r16818 r16939	287 51 18 15 23

Table A.5: Features and subsequent bugfixes on them added to improve hidden service performance

Date	Description	Revision	Size
Nov 5, 2008	When the client launches an introduction circuit, retry with a new circuit after 30 seconds rather than 60 seconds.	r17106 r17113 r17189	73 15 15
Oct 15, 2008	Launch a second client-side introduction circuit in parallel after a delay of 15 seconds.	r17108	86
Dec 10, 2008	Hidden services start out building five introduction circuits rather than three, and when the first three finish they publish a service descriptor using those.	r17110 r17111 r17562	100 24 98

Bibliography

[1] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[2] Kevin S. Bauer, Damon McCoy, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Low-resource routing attacks against Tor. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2007)*. ACM, October 2007.

[3] Oliver Berthold, Hannes Federrath, and Stefan Köpsell. Web MIXes: A system for anonymous and unobservable Internet access. In Hannes Federrath, editor, *Proceedings of International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 115–129. Springer, July 2000.

[4] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of service or denial of security? How attacks on reliability can compromise anonymity. In *Proceedings of CCS 2007*, October 2007.

[5] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2004)*, pages 77–84. ACM, October 2004.

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

[7] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, 4th edition, June 2005.

[8] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *Proceedings of the Symposium on Security and Privacy (S&P 2003)*, pages 2–15. IEEE Computer Society, May 2003.

[9] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol—version 1.1. Request for Comments 4346, April 2006. `http://www.ietf.org/rfc/rfc4346.txt` (last checked: Dec 6, 2008).

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

[11] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320. USENIX, August 2004.

[12] Markus Dorsch, Martin Grote, Knut Hildebrandt, Maximilian Röglinger, Matthias Sehr, Christian Wilms, Karsten Loesing, and Guido Wirtz. Concealing presence information in instant messaging systems—protocol specification. Technical Report 66, Otto-Friedrich-Universität Bamberg, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik, April 2006.

[13] John R. Douceur. The sybil attack. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, volume 2429 of *Lecture Notes in Computer Science*. Springer, March 2002.

[14] Morris Dworkin. Recommendation for block cipher modes of operation, methods and techniques. Technical Report 800-38A, National Institute of Standards and Technology, December 2001.

[15] Nicholas A. Fraser, Douglas J. Kelly, Richard A. Raines, Rusty O. Baldwin, and Barry E. Mullins. Using client puzzles to mitigate distributed denial of service attacks in the Tor anonymous routing environment. In *Proceedings of the International Conference on Communications (ICC 2007)*, pages 1197–1202. IEEE Computer Society, June 2007.

[16] Michael J. Freedman and Robert Morris. Tarzan: a peer-to-peer anonymizing network layer. In *Proceedings of the 9th Conference on Computer and Communications Security (CCS 2002)*, pages 193–206. ACM, November 2002.

[17] Michael J. Freedman, Emil Sit, Josh Cates, and Robert Morris. Introducing Tarzan, a peer-to-peer anonymizing network layer. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, volume 2429 of *Lecture Notes in Computer Science*, pages 121–129. Springer, March 2002.

[18] *The Gnutella Protocol Specification v0.4*. `http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf` (last checked: Dec 6, 2008).

[19] Ian Goldberg. *A Pseudonymous Communications Infrastructure for the Internet*. PhD thesis, University of California at Berkeley, December 2000.

[20] Ian Goldberg. Privacy-enhancing technologies for the Internet, II: Five years later. In Roger Dingledine and Paul Syverson, editors, *Proceedings of the Second Workshop on Privacy Enhancing Technologies*

(PET 2002), volume 2482 of *Lecture Notes in Computer Science*, pages 1–12. Springer, April 2002.

[21] Ian Goldberg. On the security of the Tor authentication protocol. In George Danezis and Philippe Golle, editors, *Proceedings of the Sixth Workshop on Privacy Enhancing Technologies (PET 2006)*, volume 4258 of *Lecture Notes in Computer Science*, pages 316–331. Springer, June 2006.

[22] Ian Goldberg. Privacy enhancing technologies for the Internet III: Ten years later. In Alessandro Acquisti, Stefanos Gritzalis, Costos Lambrinoudakis, and Sabrina di Vimercati, editors, *Digital Privacy: Theory, Technologies, and Practices*, chapter 1, pages 3–18. Auerbach, December 2007.

[23] Ian Goldberg and David Wagner. TAZ servers and the rewebber network: Enabling anonymous publishing on the world wide web. *First Monday*, 3(4), August 1998.

[24] Ian Goldberg, David Wagner, and Eric Brewer. Privacy-enhancing technologies for the Internet. In *Proceedings of the 42nd IEEE Spring COMPCON*. IEEE Computer Society, February 1997.

[25] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Hiding routing information. In Ross J. Anderson, editor, *Proceedings of Information Hiding: First International Workshop*, volume 1174 of *Lecture Notes in Computer Science*, pages 137–150. Springer, May 1996.

[26] Saikat Guha and Paul Francis. Identity trail: Covert surveillance using DNS. In Nikita Borisov and Philippe Golle, editors, *Proceedings of the Seventh Symposium on Privacy Enhancing Technologies (PET 2007)*, volume 4776 of *Lecture Notes in Computer Science*, pages 153–166. Springer, June 2007.

[27] Ceki Gülcü and Gene Tsudik. Mixing e-mail with babel. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 1996)*, pages 2–16. Internet Society, February 1996.

[28] Knut Hildebrandt. Konzeption von Authentifizierungsmechanismen für anonyme Dienste und Realisierung eines ausgewählten Verfahrens im Anonymisierungsnetzwerk Tor. Diploma thesis, Otto-Friedrich-Universität Bamberg, January 2007.

[29] Simon Josefsson. The base16, base32, and base64 data encodings. Request for Comments 4648, October 2006. `http://www.ietf.org/rfc/rfc4648.txt` (last checked: Dec 6, 2008).

[30] Sven Kaffille, Karsten Loesing, and Guido Wirtz. Distributed service discovery with guarantees in peer-to-peer networks using distributed hashtables. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2005)*, pages 578–584, June 2005.

[31] Tobias Kamm, Thomas Lauterbach, Karsten Loesing, Ferdinand Rieger, and Christoph Weingarten. Hidden service authentication. Tor Proposal 121, The Tor Project, September 2007. `https://svn.torproject.org/svn/tor/trunk/doc/spec/proposals/121-hidden-service-authentication.txt` (last checked: Dec 6, 2008).

[32] Aniket Kate, Greg Zaverucha, and Ian Goldberg. Pairing-based onion routing. In Nikita Borisov and Philippe Golle, editors, *Proceedings of the Seventh Symposium on Privacy Enhancing Technologies (PET 2007)*, volume 4776 of *Lecture Notes in Computer Science*. Springer, June 2007.

[33] Stefan Köpsell. Low latency anonymous communication—How long are users willing to wait? In Günter Müller, editor, *Emerging Trends*

in Information and Communication Security (ETRICS 2006), volume 3995 of *Lecture Notes in Computer Science*, pages 221–237. Springer, June 2006.

[34] Jim F. Kurose and Keith W. Ross. *Computer Networks—A Top-Down Approach*. Addison-Wesley, 4th edition, April 2007.

[35] Jörg Lenhard. Anonymous access to public services in the Internet—analysis of Tor hidden services in low bandwidth networks. Bachelor thesis, Otto-Friedrich-Universität Bamberg, December 2008.

[36] Karsten Loesing. Distributed storage for Tor hidden service descriptors. Tor Proposal 114, The Tor Project, May 2007. `https://svn.torproject.org/svn/tor/trunk/doc/spec/proposals/114-distributed-storage.txt` (last checked: Dec 6, 2008).

[37] Karsten Loesing. Improvements of distributed storage for Tor hidden service descriptors. Tor Proposal 143, The Tor Project, June 2008. `https://svn.torproject.org/svn/tor/trunk/doc/spec/proposals/143-distributed-storage-improvements.txt` (last checked: Dec 6, 2008).

[38] Karsten Loesing. Simplify configuration of private Tor networks. Tor Proposal 135, The Tor Project, April 2008. `https://svn.torproject.org/svn/tor/trunk/doc/spec/proposals/135-private-tor-networks.txt` (last checked: Dec 6, 2008).

[39] Karsten Loesing, Markus Dorsch, Martin Grote, Knut Hildebrandt, Maximilian Röglinger, Matthias Sehr, Christian Wilms, and Guido Wirtz. Privacy-aware presence management in instant messaging systems. In *Proceedings of the 20th International Parallel and Dis-*

tributed Processing Symposium (IPDPS 2006). IEEE Computer Society, April 2006.

[40] Karsten Loesing, Maximilian Röglinger, Christian Wilms, and Guido Wirtz. Implementation of an instant messaging system with focus on protection of user presence. In *Proceedings of the Second International Conference on Communication System Software and Middleware (COMSWARE 2007)*. IEEE Computer Society, January 2007.

[41] Karsten Loesing, Werner Sandmann, Christian Wilms, and Guido Wirtz. Performance measurements and statistics of Tor hidden services. In *Proceedings of the International Symposium on Applications and the Internet (SAINT 2008)*, Turku, Finland, July 2008. IEEE Computer Society.

[42] Karsten Loesing and Christian Wilms. Combine introduction and rendezvous points. Tor Proposal 142, The Tor Project, June 2008. `https://svn.torproject.org/svn/tor/trunk/doc/spec/proposals/142-combine-intro-and-rend-points.txt` (last checked: Dec 6, 2008).

[43] Karsten Loesing and Christian Wilms. Four improvements of hidden service performance. Tor Proposal 155, The Tor Project, September 2008. `https://svn.torproject.org/svn/tor/trunk/doc/spec/proposals/155-four-hidden-service-improvements.txt` (last checked: Dec 6, 2008).

[44] Karsten Loesing and Guido Wirtz. An implementation of reliable group communication based on the peer-to-peer network JXTA. In *Proceedings of the International Conference on Computer Systems and Applications (AICCSA 2005)*. IEEE Computer Society, January 2005.

[45] Karsten Loesing and Guido Wirtz. Virtual private services. Technical report, HotPETs Session on the Eighth Symposium on Privacy

Enhancing Technologies (HOT-PETs 2008), Leuven, Belgium, July 2008. `http://petsymposium.org/2008/hotpets/vrtprsvc.pdf` (last checked: Dec 6, 2008).

[46] Nick Mathewson and Roger Dingledine. Mixminion: Strong anonymity for financial cryptography. In Ari Juels, editor, *Proceedings of the 8th International Conference on Financial Cryptography (FC 2004)*, volume 3110 of *Lecture Notes in Computer Science*. Springer, February 2004.

[47] David Mazières and M. Frans Kaashoek. The design, implementation and operation of an email pseudonym server. In *Proceedings of the 5th Conference on Computer and Communications Security (CCS 1998)*. ACM, November 1998.

[48] Damon McCoy, Kevin Bauer, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Shining light in dark places: Understanding the Tor network. In Nikita Borisov and Ian Goldberg, editors, *Proceedings of the Eighth Symposium on Privacy Enhancing Technologies (PETS 2008)*, volume 5134 of *Lecture Notes in Computer Science*, pages 63–76, Leuven, Belgium, July 2008. Springer.

[49] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC, December 1996.

[50] Ulf Möller, Lance Cottrell, Peter Palfrader, and Len Sassaman. Mixmaster protocol version 2. IETF Internet Draft, July 2003.

[51] Steven J. Murdoch. Hot or not: Revealing hidden services by their clock skew. In *Proceedings of the 13th Conference on Computer and Communications Security (CCS 2006)*. ACM, October 2006.

[52] Steven J. Murdoch. *Covert channel vulnerabilities in anonymity systems*. PhD thesis, University of Cambridge, December 2007.

[53] Steven J. Murdoch and Robert N. M. Watson. Metrics for security and performance in low-latency anonymity networks. In Nikita Borisov and Ian Goldberg, editors, *Proceedings of the Eigth Symposium on Privacy Enhancing Technologies (PETS 2008)*, volume 5134 of *Lecture Notes in Computer Science*, pages 115–132. Springer, July 2008.

[54] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*, November 2001. Federal Information Processing Standards Publication 197.

[55] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*, October 2008. Federal Information Processing Standards Publication 180-3.

[56] Lasse Øverlier and Paul Syverson. Locating hidden servers. In *Proceedings of the Symposium on Security and Privacy (S&P 2006)*. IEEE Computer Society, May 2006.

[57] Lasse Øverlier and Paul Syverson. Valet services: Improving hidden servers with a personal touch. In George Danezis and Philippe Golle, editors, *Proceedings of the Sixth Workshop on Privacy Enhancing Technologies (PET 2006)*, volume 4258 of *Lecture Notes in Computer Science*, pages 223–244. Springer, June 2006.

[58] Lasse Øverlier and Paul Syverson. Improving efficiency and simplicity of Tor circuit establishment and hidden services. In Nikita Borisov and Philippe Golle, editors, *Proceedings of the Seventh Symposium on Privacy Enhancing Technologies (PET 2007)*, volume 4776 of *Lecture Notes in Computer Science*, pages 134–152. Springer, June 2007.

[59] Andriy Panchenko, Lexi Pimenidis, and Johannes Renner. Performance analysis of anonymous communication channels provided by Tor. In *Proceedings of the Third International Conference on Availability,*

Reliability and Security (ARES 2008), pages 221–228. IEEE Computer Society, March 2008.

[60] Sameer Parekh. Prospects for remailers—where is anonymity heading on the internet? *First Monday*, 1(2), August 1996.

[61] Andreas Pfitzmann and Marit Hansen. Anonymity, unlinkability, unobservability, pseudonymity, and identity management—a consolidated proposal for terminology. `http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.31.pdf` (last checked: Dec 6, 2008), February 2008.

[62] Andreas Pfitzmann, Birgit Pfitzmann, and Michael Waidner. ISDN-mixes: Untraceable communication with very small bandwidth overhead. In *Proceedings of the GI/ITG Conference on Communication in Distributed Systems*, volume 267 of *Informatik-Fachberichte*, pages 451–463. Springer, February 1991.

[63] The Tor Project. *Tor directory protocol, version 3*, 2008. `https://svn.torproject.org/svn/tor/trunk/doc/spec/dir-spec.txt` (last checked: Dec 6, 2008).

[64] The Tor Project. *Tor Protocol Specification*, 2008. `https://svn.torproject.org/svn/tor/trunk/doc/spec/tor-spec.txt` (last checked: Dec 6, 2008).

[65] The Tor Project. *Tor Rendezvous Specification*, 2008. `https://svn.torproject.org/svn/tor/trunk/doc/spec/rend-spec.txt` (last checked: Dec 6, 2008).

[66] Jean-François Raymond. Traffic analysis: Protocols, attacks, design issues, and open problems. In Hannes Federrath, editor, *Proceedings of International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 10–29. Springer, July 2000.

[67] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.

[68] Michael Reiter and Aviel Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, June 1998.

[69] Ron L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[70] Len Sassaman, Bram Cohen, and Nick Mathewson. The pynchon gate: A secure method of pseudonymous mail retrieval. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2005)*. ACM, November 2005.

[71] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, January 1996.

[72] Andrei Serjantov and Peter Sewell. Passive attack analysis for connection-based anonymity systems. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS 2003)*, volume 2808 of *Lecture Notes in Computer Science*. Springer, October 2003.

[73] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, volume 2429 of *Lecture Notes in Computer Science*. Springer, March 2002.

[74] Robin Snader and Nikita Borisov. A tune-up for Tor: Improving security and performance in the Tor network. In *Proceedings of the Network*

and Distributed System Security Symposium (NDSS 2008). Internet Society, February 2008.

[75] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the Conference of the Special Interest Group on Data Communication (SIGCOMM 2001)*, pages 149–160. ACM, August 2001.

[76] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, February 2003.

[77] Paul Syverson, Gene Tsudik, Michael Reed, and Carl Landwehr. Towards an analysis of onion routing security. In Hannes Federrath, editor, *Proceedings of International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 96–114. Springer, July 2000.

[78] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems—Principles and Paradigms*. Prentice-Hall, February 2002.

[79] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2008. `http://cran.r-project.org/doc/manuals/refman.pdf` (last checked: Dec 6, 2008).

[80] Rolf Wendolsky, Dominik Herrmann, and Hannes Federrath. Performance comparison of low-latency anonymisation services from a user perspective. In Nikita Borisov and Philippe Golle, editors, *Proceedings of the Seventh Symposium on Privacy Enhancing Technologies*

(PET 2007), volume 4776 of *Lecture Notes in Computer Science*, pages 233–253. Springer, June 2007.

[81] Christian Wilms. Improving the Tor hidden service protocol aiming at better performance. Diploma thesis, Otto-Friedrich-Universität Bamberg, June 2008.

[82] Christian Wilms. Improving the Tor hidden service protocol aiming at better performance. Technical Report 79, Otto-Friedrich-Universität Bamberg, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik, November 2008.

[83] Matthew Wright, Micah Adler, Brian Neil Levine, and Clay Shields. An analysis of the degradation of anonymous protocols. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2002)*. Internet Society, February 2002.

[84] Matthew Wright, Micah Adler, Brian Neil Levine, and Clay Shields. Defending anonymous communication against passive logging attacks. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P 2003)*, pages 28–43. IEEE Computer Society, May 2003.

[85] Matthew Wright, Micah Adler, Brian Neil Levine, and Clay Shields. The predecessor attack: An analysis of a threat to anonymous communications systems. *ACM Transactions on Information and System Security (TISSEC)*, 4(7):489–522, November 2004.

[86] Sebastian Zander and Steven J. Murdoch. An improved clock-skew measurement technique for revealing hidden services. In *Proceedings of the 17th USENIX Security Symposium*. USENIX, July 2008.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne die Hilfe eines Promotionsberaters angefertigt habe. Dabei habe ich keine anderen Hilfsmittel als die im Literaturverzeichnis genannten benutzt. Alle aus der Literatur wörtlich oder sinngemäß entnommenen Stellen sind als solche kenntlich gemacht.

Weder diese Arbeit noch wesentliche Teile derselben wurden einer anderen Prüfungsbehörde zur Erlangung des Doktorgrades vorgelegt.

Die Arbeit wurde bisher noch nicht in ihrer Ganzheit publiziert. Alle bereits veröffentlichten Beiträge, auf denen diese Arbeit basiert, sind im Literaturverzeichnis unter den Nummern [12, 30, 31, 36–45] angegeben.

UNIVERSITY OF BAMBERG PRESS

Privatsphäre im Internet wird immer wichtiger, da ein zunehmender Teil des alltäglichen Lebens über das Internet stattfindet. Internet-Benutzer verlieren die Fähigkeit zu steuern, welche Informationen sie über sich weitergeben oder wissen nicht einmal, dass sie dieses tun. Datenschutzfördernde Techniken helfen dabei, private Informationen im Internet zu kontrollieren, zum Beispiel durch die Anonymisierung von Internetkommunikation. Bis heute liegt der Hauptfokus dieser Techniken auf dem Schutz des Anfragenden beim Zugriff auf öffentliche Dienste. Nur ein Teilbereich von datenschutzfördernden Techniken, die sogenannten pseudonyme Dienste, zielt darauf ab, den Standort eines Servers und damit eines Dienstanbieters zu verbergen. Diese Arbeit wirft die Frage nach den Risiken beim Betrieb von Internetdiensten durch Privatpersonen auf und schlägt die Nutzung und Erweiterung von pseudonymen Diensten für private Dienstanbieter vor.

Die Arbeit besteht aus drei Hauptbeiträgen: Erstens wird vorgeschlagen, Dienstbeschreibungen von pseudonymen Diensten in geeigneter Weise in einer verteilten Datenstruktur abzulegen, um eine höhere Skalierbarkeit zu erreichen. Zweitens werden zwei Vorschläge gemacht, wie die Informationen, die ein Dienst im Netzwerk bekanntgibt, auf ein Minimum reduziert und nicht-autorisierte Clients schon während der Verbindungsherstellung am Zugriff gehindert werden können. Drittens wird die Effizienz des Anbietens und Zugreifens von pseudonymen Dienste gemessen, um mögliche Engpässe zu identifizieren und verbessern zu können.

ISBN 978-3-923507-45-0
ISSN 1867-7401
17,60 Euro