Høgskolen i Gjøviks rapportserie, 2015 nr. 1

Past, Present and Future of Tor Hidden Services

Konstantin Müller

Høgskolen i Gjøvik
2015

ISSN: 1890-520X

ISBN: 978-82-93269-89-2

Past, Present and Future of Tor Hidden Services

Konstantin Müller

Abstract

Using the Internet reveals the IP addresses of both communication parties to everybody, who is able to observe the communication. Anonymity systems like Tor hide the IP addresses and the communication in such a way that the communication cannot be linked between the two parties. Tor is a low-latency anonymity network designed for interactive applications, which allows its users to stay anonymous while using the Internet, for example for Web browsing, e-mail or instant messaging. In order to achieve anonymity the users' traffic is routed through the Tor network by choosing three random network nodes. Thus, the user stays anonymous, because his real IP address is not used to access the service and no single node in the network can link the user to the service he is accessing. In addition, Tor provides *location-hidden services* to allow services to operate anonymous as well. With location-hidden services the location of a service, i.e. its IP address, is not revealed by using the service. At the moment the Tor Project works on an overhaul of the design of location-hidden services.

This paper introduces Tor, explains its design and specification with the goal to understand, how Tor is working to achieve its anonymity goals. It explains why Tor is built in the way as it is implemented today, but also states Tor's limitations and situations in which Tor cannot guarantee anonymity. Furthermore, location-hidden services are introduced as well. The current hidden services design is examined in-depth with its shortcomings and drawbacks and it is presented why this design needs to be renewed. This gives a rationale for a new design of hidden services. The proposed new design is described as well. The paper finishes with a summary of open research questions, which are not yet addressed by the new hidden services design.

The goal of this paper is to provide the reader with a thorough understanding of Tor and its location-hidden services. It goes beyond most introductions of Tor as it looks into Tor's protocol specifications and explains the choices made by the designers and developers of Tor. After reading this paper the reader should be able explain, how Tor is built, why it is implemented this way, but also what Tor's limitations are regarding anonymity and what aspects of Tor are not yet well understood. The focus of this paper is laid on the main Tor application and anonymity. It does not consider application-level issues such as the *Tor Browser* or the usage of Tor as a censorship circumvention tool.

Contents

List of Figures

1 Introduction

People want to be anonymous on the Internet for various reasons. They do not want that others can get to know, what Web sites they browse, for example if they search for health related topics or other sensitive information. Or they simply do not want to be tracked by advertising companies. Activists may come under repression, if their activities will be revealed. Companies do not want that their competitors know, what they are doing on the Internet. Law enforcement has good operational reasons to be anonymous while investigating crime online. The military does not want to reveal the location of their units while communicating over the Internet. All these are good reasons for individuals and organisation to communicate anonymously over the Internet.

Tor is a low-latency anonymity system [1], which allows such anonymous communication. It is designed for interactive applications such as Web browsing, remote access or instant messaging and can handle any TCP traffic. The Tor network consists of relays run by volunteers, which transport traffic through and out the Tor network. Clients connect to the network, choose randomly three different relays to form a path through the network and send their requests over this path. The last relay in the path connects to the user's destination, for example a Web site, and sends the response back to the user. Because every relay only knows its predecessor and successor on the path, no single relay can link the user with the service he accessed, thus, preserving the user's anonymity.
As of the time of writing the Tor network consists of about 6000 relays, which transports roughly 5000 MiB/s of data. The Tor Project estimates that about over 2.5 million clients connect to the network everyday[1].

Tor does not only provide anonymity to its users, but can also help services operators to run their services anonymously with *location-hidden services*. A location-hidden service, for instance a Web site, connects to the Tor network similarly as an ordinary Tor user and waits for users to connect to it. People who want to access this service use Tor normally and the Tor network handles the connection between the user and the hidden service. With this mechanism users can access the service anonymously. At the same time the services can be offered anonymous as well. The IP addresses of both the user and service are not revealed.
Location-hidden services allows, for example, human rights activists to publish information about violations of human rights in a safe and anonymous way. Without hidden services this information could be easily censored or the service shut down. Also the publisher could face serious consequences for publishing, if the information could be attributed to him.

[1] All statistics about the Tor network can be found at `https://metrics.torproject.org/`.

1.1 Motivation and Goals

This paper introduces Tor with a special focus on location-hidden services. It explains what kind of threats Tor wants to protect against and what design goals Tor has. The design and specification of Tor is analysed with regard to how Tor tries to achieve these goals and how Tor maintains the user's and service operator's anonymity. This includes limitations and shortcomings of Tor and describes situations in which Tor cannot by design guarantee anonymity and situations in which Tor could be improved to provide better anonymity.
The Tor Project is working on a renewal of location-hidden services in order to improve the current hidden services design. The current design is examined in-depth and analysed what drawbacks this design has and why a renewal is necessary. Keeping this in mind the proposed new hidden service design is described as well and it is explained how the new design attempts to fix the problems of the old one. This includes open research questions, which are not yet addressed by the new design and need to be studied before it can be implemented.

The goal of this paper is to provide the reader with a thorough understanding of Tor and location-hidden services. It goes beyond most introductions to Tor as it looks into the specifications of Tor and analyses why Tor is built in the way as it is implemented today. This gives the reader a deep understanding of Tor, its design and specification. But at the same time Tor's limitations and shortcomings are not left out, such that the reader should be able to understand in which situations Tor cannot guarantee anonymity. This is important, because using Tor in situations, which give a false sense of anonymity, could possibly have serious consequences.
This paper concentrates on the main Tor network and how to protect anonymity with a special focus on location-hidden services. Application-specific issues such as protecting anonymity at the application level are not considered in this paper as well as using Tor for censorship circumvention or solving the problem of companies and countries trying to prevent the usage of Tor. The following related work section lists some reference the reader can consult, if he is interested in these topics.

The rest of this paper is organised in the following way. The next section lists related work with regard to the topic of this paper, but also provides further references, which can be read to learn more about Tor and Tor related topics not covered in this paper. Chapter 2 first gives a short introduction into the problem of anonymity itself, because anonymity has quite different security requirements than the typical CIA-triplet confidentiality, integrity and availability. Further is the main Tor network described and explained in detail how Tor protects the user's anonymity. Chapter 3 starts with an analysis of the current hidden service design, which leads to the proposed new hidden service design, and ends with open research questions regarding this proposal. Chapter 4 summarises and concludes this paper.

1.2 Related Work

The original Tor design is introduced in [1]. It describes the threat model Tor tries to protect against and the design goals behind Tor, which gives a rationale for the actual design and implementation of Tor. [2] follows up on the original design paper and reports on challenges in

deploying Tor and the difficulties of establishing a low-latency anonymity system. The design of Tor as a blocking-resistant anonymity system, which can be used for censorship circumvention, is described in [3]. The design of the *Tor Browser*, which is a Web browser based on Tor and Firefox optimised for anonymity and fixing many application-level issues regarding anonymity and Web browsing, is explained in [4]. In general, over the years since Tor's invention the Tor Project released many technical reports about various parts of the Tor ecosystem, which are all published at `https://research.torproject.org/techreports.html`. In addition, all the Tor specifications are publicly available at `https://gitweb.torproject.org/torspec.git`.

Over the years many researchers tried to attack Tor. For example, [5] identifies the path of a Tor connection by measuring difference in the latency of relays, when the traffic of a connection is going through a relay. [6] analyses the probability of choosing a compromised path if an attacker is able to observe or control a large fraction of the Tor network. In [7] the first attack against hidden services is presented, which is able to find the IP address of a hidden service by just running one single relay. An attack against hidden services, which exploits the fact that the clock skew of a computer changes under load due to an increased CPU temperature, is explained in [8]. With this attack a traffic pattern is sent to a hidden service and at the same time the clock skew of potential hidden services measured directly outside the Tor network. If the traffic pattern and the measured pattern of the clock skew matches, the hidden service is deanonymised. An improved version of this attack is described in [9]. Further attacks against hidden services, which also motivated a redesign of hidden services, are presented in [10]. Some of the attacks against hidden services are explained in more detail in Chapter 3.

Some efforts were made to measure the performance of hidden services in different scenarios [11, 12] and to improve the performance [13]. The latter paper explains different possibilities to optimise the path creation. Further improvements of hidden services are proposed in [14]. The presented design of *Valet nodes* improves the availability and resistance against Denial-of-Service attacks of hidden services. Additionally, it allows *completely hidden services* in the sense, that even the Tor network itself does not know the existence of a hidden service. [15] lists different ways to deal with botnets, which utilise hidden services for their purposes.

In order to understand the design of Tor it is also important to understand anonymity research in general. For instance, [16] models the usage of anonymity networks from an economics and game theory perspective. When do people use an anonymity network and when not? What are incentives to voluntarily run relays in such a network? In addition, [17] explains why usability is important for an anonymity network and why a high secure system can actually hurt the user's anonymity. In general, the reader, who is interested in Tor and anonymity research, should visit the *Anonbib* at `http://freehaven.net/anonbib/topic.html`, which contains a large list of research papers about anonymity systems, including Tor related papers. A definitively worthwhile read is [18], which proposed the first anonymity system aimed at e-mails and developed the fundamental ideas of modern anonymity research.

2 The Second-Generation Onion Router

The Internet was not designed and built with security in mind. Instead secure communication protocols were developed on top of this insecure foundation. Traditionally, the three considered aspects of information security are *confidentiality*, *integrity* and *availability*. Confidentiality prevents the "unauthorized disclosure", integrity the "unauthorized modification" and availability the "unauthorized withholding" of information [19, p. 34]. Different application areas add further aspects to this basic CIA-triplet, with anonymity being one of them. This paper uses the definition of anonymity as given by Gollmann:

> **Anonymity** – A subject (user) is *anonymous* if it cannot be identified within a given *anonymity set* of subjects. [19, p. 35]

Confidentiality can be achieved with the encryption of information, such that only the users in possession of the correct key can decrypt and read the encrypted information. This hides the information itself, but does not hide who is communicating with whom. The latter is the goal of anonymity. People communicate with each other without revealing the existence of this communication and the communication partners.

There exists a fundamental difference between confidentiality and anonymity. Confidentiality can be achieved by the communication partners alone, anonymity not. For reaching anonymity users must blend in with a group of other users, the anonymity set from the definition above. Different users communicate over the same anonymity network and provide *cover traffic* for the other users. If the users in the anonymity set and their communications are undistinguishable, an attacker cannot figure out, who is communicating with whom, thus, the users are anonymous inside the given anonymity set.

In general, anonymity systems can be divided into two different main categories: *high-latency* and *low-latency* anonymity systems [17]. The goal of high-latency systems is to protect against strong global attackers, which are able to observe a large fraction of the network to conduct *traffic correlation attacks*. Such an attack compares traffic patterns such as timing or volume characteristics between communication parties in order to link traffic between the parties and revealing, who communicates with whom. High-latency systems protect against such attacks by destroying the patterns, for example by introducing random delays before messages are forwarded inside the network. Such a system offers higher security, but is inappropriate for interactive applications, because users expect a quick response from these applications.

In contrast, low-latency anonymity systems have a weaker threat model by not necessarily directly protecting against traffic correlation attacks. Because of that, such a system is suitable for a lot more applications such as Web browsing or instant messaging and offers a better usability to

its users. This can in fact lead to a higher degree of anonymity compared to high-latency systems as shown in the following section. Tor is the most widely-used implementation of a low-latency anonymity system today.

2.1 Anonymity Loves Company

The previous section explains that an anonymity system needs users, who provide cover traffic to other users. In general, a larger anonymity set can provide better anonymity, because it will be harder for an attacker to link network activities to specific users. Thus, an anonymity system needs to attract as much users as possible. But every user has different requirements with regard to both security and anonymity, but also in the way he is using the anonymity system. [16] distinguishes between low-sensitivity users and high-sensitivity users. High-sensitivity users have higher requirements of security and anonymity as low-sensitivity users. They may choose to use a system A, which is highly secure but has a lower performance and is only useful for a handful of applications. But this system has a small anonymity set consisting only of high-sensitivity users, thus, it cannot provide very much anonymity. In contrast, an anonymity system B with a lot of low-sensitivity users can offer stronger anonymity and resistance against attacks even if its design is not as secure as the design of system A.

The above highlights that sometimes it may be better for anonymity to have more users and a weaker anonymity system than a strong system and only a few users [17]. Additionally, it shows that usability and performance of an anonymity system are equally important than security and anonymity properties. An anonymity systems needs to attract low-sensitivity users in order to provide cover traffic for high-sensitivity users [16]. If these users turn away because of bad usability, they will not have anonymity at all. At the same time high-sensitivity users are put at risk of deanonymisation, because they lost their cover traffic to hide in. When designing an anonymity system this insight must be always kept in mind. A secure system alone is not enough, usability is a key factor to attract enough users [17]. The difficulty is to build an anonymity system, which is secure on one hand but also usable on the other hand.

But the size of the anonymity set alone is not enough. For example, if a company uses its own anonymity system in order to hide its communications from its competitors, this system does not provide any anonymity at all. Every communication will be by definition originate from this company. Because of that, user diversity is important as well. An anonymity system needs to attract users with different backgrounds, objectives, motivations and reasons for using the system in order to make it harder for an attacker to deanonymise users based on these properties. In this context the authors of the Crowds anonymity network [20] coined the fitting phrase “anonymity loves company”.

Another obstacle needs to be solved in order to use an anonymity system in a secure way. The behaviour of users must be indistinguishable, i.e. users must act in the same way. For instance, a user who behaves completely different than the other users stands out from them and is therefore easily recognisable. The same holds also for the client software utilised to access the anonymity

network. If the client software behaves differently compared to other users' software, this can be used by an attacker to identify the user. The reason could be misconfiguration or unusual settings. A user changing the settings with the goal to gain more security can actually hurt his anonymity, because the software behaves different compared to the users without changed settings. Thus, different users may make other security-anonymity trade-offs based on their personal requirements. This demonstrates again that anonymity set size and user diversity is important, because it is more difficult with a larger set and more diversity to find users who stand out. On the other hand it may be possible to partition users in various groups based on their behaviour or client configuration, which makes attacks to identify users easier, because an attacker can concentrate on only one smaller group of users.

This section shows that designing an anonymity system is a very difficult task. Trade-offs between anonymity, security and usability must be carefully examined and balanced. Focusing to close on only one property could have a negative impact on the other properties. It is important to understand this set of problems, because the design of Tor is motivated by them and it helps to comprehend it.

2.2 Introduction

Over the years different designs for anonymity systems were proposed. One of them is *onion routing* and Tor is the "second-generation onion router" [1], which improves the original onion routing design. Messages are wrapped in multiple layers of encryption and sent through the network until its final destination. The Tor network consists of relays or nodes run by volunteers around the world, which in the Tor terminology are called *onion router* (OR). Each user runs a client software, named *onion proxy* (OP), in order to connect to the network. The OP exposes a SOCKS proxy [21], which provides a common interface to applications in order to utilise Tor for anonymous communication. With this approach Tor is able to transport arbitrary TCP streams without having to deal with application- or protocol-specific properties. Additionally, users do not need to modify their applications, for instance a browser can be easily configured to use Tor as a proxy server. Tor does not support UDP.

2.2.1 Design Goals and Adversary

Based on previous experiences with a first-generation onion routing prototype the Tor designers improved the original design with a few design goals in mind: *deployability*, *usability*, *flexibility* and *simple design*. These goals reflect the results of general anonymity research as presented in Section 2.1. Having a system, which is easy to deploy, lowers the barrier to become a node operator. More nodes are important to create more diversity and to increase the overall performance of the network. It also makes it harder for an attacker, because it becomes more expensive to observe a large fraction of the network. As explained usability is a key factor to attract users, which provide cover traffic to other users. A flexible and simple design facilitates quicker response to attacks against the system or to new system requirements, which needs a system redesign.

Furthermore, Tor has a few non-goals as well partly reflecting the overall goals. Tor does not

use a pure peer-to-peer design with short-lived user nodes. Instead it is built on top of a simpler design with voluntarily run relays. As a low-latency anonymity network Tor does not explicitly protect against traffic correlation attacks, because no usable solution for low-latency networks is known yet [1]. Tor also does not perform protocol normalisation, i.e. the removal of identifying information from higher level protocols like HTTP. For this task Tor can be combined with specialised proxy software layered between Tor and the application. With the initial design Tor is not trying to conceal that someone is using Tor. This changed partly with the introduction of a blocking-resistant Tor design [3] used for censorship circumvention. But still today the main Tor network does not use steganography.

In contrast to high-latency anonymity networks Tor does not try to protect against a global passive attacker, who is able to observe a large fraction or even the whole network. Such an attacker is in the position to effectively conduct traffic correlation attacks. Tor assumes a weaker attacker, which can observe or compromise only a part of the whole network. This kind of attacker must either control the first and last node on a path or the connection between the user and the first node and the connection between the last node and the destination [6]. Because of this, Tor aims to make it difficult for an attacker to come into the position of observing initiator and responder of a communication. An attacker may be able to observe or compromise nodes, operate own nodes, listen on connections between nodes and between nodes and users including tampering with the data sent over those connections. He may try to block the usage of the network or give users a false list of network nodes. See [1] for a list of attacks against Tor and for an analysis how well Tor protects against these attacks.
Attackers may want to identify the communication partners, link communication to a user, for example to determine that Alice viewed Web site XYZ, or make a profile of a user. A profile does not need to be attributed to the identity of a user. It may be enough for an attacker to know what Web sites a specific but unidentified user accessed, for instance for targeted advertisements.

2.2.2 Bird's Eye View

Tor's design is based on a distributed trust model, see Figure 1 on the next page for a graphical overview of the basic mode of operation of the Tor network. A user does not need to trust a single entity in the network. Instead three different nodes are chosen to build a path through the network: an *entry node*, a *middle node* and an *exit node*. This path a is called a *circuit*. The circuit is constructed one hop at a time. First the user connects to the entry node, then extends the circuit to the middle node and the exit node. This has the advantage that in the case one node fails not the whole circuit needs to be rebuilt, instead the circuit can be extended again from the last working node. The Tor client establishes with each node session keys during this procedure. The client encrypts the data, which should be sent through the network, with each session key. The data is encrypted three times and every node on the circuit removes one layer of encryption, such that the exit node receives the plaintext. This node connects to the final destination and sends the plaintext to it, for instance connecting to a Web site and requesting a specific page. The response is transmitted back through the circuit with the nodes in reverse order. Every node encrypts the data with its session key, such that only the client can decrypt the response, because

only he knows all three session keys. This procedure of adding and removing different layers of encryption is the reason for the “onion” analogy.

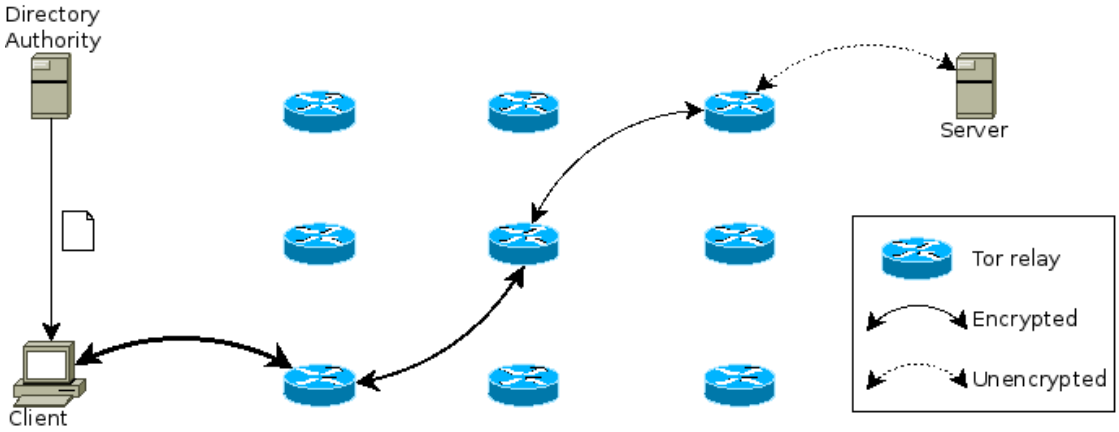

Figure 1: Overview of the Tor Network

The first node on the circuit knows the user, who is accessing Tor, but does not know the content of the data sent over the network. The exit node knows the destination of the communication and can see the data in plaintext, if no end-to-end encryption protocol like TLS for encrypted Web traffic is used. But the exit node does not know, who sent the messages. With this approach no single node alone on the circuit can link together the user with the destination of the communication. Every node only knows it predecessor and successor on the circuit. Because of that, only if an attacker controls both the entry and the exit node, he can deanonymise the user. The user data can exit at any position on the circuit, but the circuit should be at least three hops long to achieve this level of security. The circuit could be longer, but it is an open research question if a longer circuit would add enough security to justify the decreased performance. For now Tor sticks to a fixed three-hop design. New circuits are built in regular intervals of at least ten minutes and old circuits are destroyed in order to avoid that attackers can make a long lasting profile of the user by either observing or compromising the exit node.

But how do clients know the nodes inside the Tor network? Every node publishes in regular intervals a signed *relay descriptor* to a set of semi-trusted *directory authorities*. This descriptor contains all necessary information needed for establishing a connection with a node (IP address, port, public keys, etc.). Thus, every directory authority knows all or most nodes in the network. They conduct a vote with the information and agree on a *consensus* document, which contains the current view of the network of all directory authorities, i.e. the nodes currently participating the the network. The final network status document together with the relay descriptors is downloaded by clients from the authorities and is used in constructing circuits.

Relays in the network can have different roles. The roles can be either defined by the relay operators or by the directory authorities during the vote for establishing a network consensus.

The directory authorities assign flags to all relays during the vote [22]. The flags inform clients about the roles of relays. The two most important roles are entry and exit nodes. An entry node is used as the first node in a circuit to enter the network and an exit node as the last node to leave the network. The directory authorities decide when a relay should be used by clients as an entry node [23]. In contrast, relay operators set up their relays to be exit nodes. Operators may choose to not allow any traffic to leave the network at their relays or can restrict to which destinations (IP address and/or port) connections are allowed from their relays. These are called *exit policies* and are implemented in order to increase the number of relays in the network [24]. The reason for exit policies is that traffic leaving the Tor network looks like originating at the exit node. Operators may not want to deal with the hassle of running an exit node, for example managing abuse complains about traffic, which exited at their relays.
Relays can also act as *directory caches*, which download network status documents and relay descriptors from the directory authorities. They provide these documents to clients, such that clients can download these documents from the caches and not directly from the directory authorities with the goal of decreasing the load on the authorities. Furthermore, relays can be *hidden service directories*, which are responsible for storing information about hidden services to be used by clients to connect to hidden services. More details about relay roles and flags, how the are assigned and used, are given in the following sections.

2.3 Specifications

The previous section gives an introduction into Tor to provide a general understanding of Tor's design and the ideas behind Tor. Based on that, this section looks deeper into the main specifications of Tor. The goal is to provide the reader with a more thorough understanding of Tor and how it is implemented. This is important in order to comprehend, how Tor can actual provide anonymity and how Tor protects against different kinds of attacks.
First the directory authorities are explained in Subsection 2.3.1, how they form the consensus and the network status and how these documents are used for various parts of the overall Tor ecosystem. Subsection 2.3.2 describes how Tor clients choose nodes in order to build circuits. This path selection algorithm is based on the network status document produced by the directory authorities. The Subsection 2.3.3 examines in detail the communication protocol used between onion routers, and between onion routers and onion proxies. This protocol is the most critical piece of the Tor network to provide anonymity to its users.
Tor's design and implementation was improved over the years. Some parts were redesigned even more than once. This paper concentrates on the most recent version of the Tor design and does not consider outdated or obsolete parts of the specifications. This should keep the following explanations simpler and easier to understand. For all the details the reader can consult the specifications directly.

2.3.1 Directory Authorities

Directory authorities are one crucial piece of the Tor network. Clients needs to know the existing relays in the network and the status of each relay in order to be able to choose a path through the network. In addition, directory authorities collect and store anonymous statistics about the usage

of the Tor network [25] and make them available to the public, where they are used inter alia for research about Tor. Currently, there are nine directory authorities, which are administrated by trusted members of the Tor community. A fixed list of all directory authorities is shipped with the Tor software in order to make it possible for clients to bootstrap the network. For a client this is enough, because he can fetch all necessary information about the Tor network from the authorities in order to start using Tor.
The specification of directory authorities, how they interact with each other, with relays and with clients, is described in [22]. This contains everything which is needed to form network status documents, to enable relays to contribute to the network and clients to use it. Without directory authorities the network as a whole could not function.

Introduction

All directory authorities have a long-term *authority identity key*, which uniquely identifies each directory authority. This key is used to sign a *key certificate* containing an *authority signing key*. The authority signing key is a medium-term key valid for 3 to 12 month and signs all documents produced by the corresponding directory authority. The authority identity key is the "Holy Grail" of any directory authority. If an attacker can compromise this key, he can successfully impersonate the authority. For this reason the authority identity key is stored encrypted and/or offline and is only needed to sign a new authority signing key, when the old key is rotated. The rotation of the authority signing key is much easier. Rotation of the authority identity key would require a cumbersome redistribution of the Tor software, because the public component of the authority identity key is hard-coded in the Tor source code. This component is needed to verify the key certificate, such that the currently used authority signing key actually belongs to the directory authority. Thus, clients can be sure that all documents signed by the authority signing key are produced by the rightful directory authority.

Every relay produces *relay descriptors* and *extra info documents* and uploads them to the directory authorities. Relay descriptors contain everything, which clients need to know about a relay in order to choose relays for a circuit and to interact with them. Extra info documents consists of additional information not strictly necessary for the operation of the Tor network itself, mainly usage statistics about the relay. The extra info documents are split from the relay descriptors, because they are not needed by clients, such that they only need to download the smaller relay descriptors.

Each directory authority knows all or at least most relays inside the Tor network. Periodically each authority forms an opinion about the network status in form of a *status vote*, which includes the relays the authority knows about and the relays' statuses. The authorities exchange their votes and each authority independently constructs a *consensus* about the current network status based on all votes. After that process all directory authorities have the same view of the network, which is signed by all authorities and distributed to clients as *network status documents*. The idea is that the authorities agree on a common view of the network, because not all authorities may have the same view. For example, a relay may not send relay descriptors to all directory authorities or the upload failed due to network timeouts during transmission. Furthermore, a

malicious directory authority alone cannot fake network status documents and distribute false information to clients.
Every network status document has three timestamps: *valid-after* (VA), *fresh-until* (FU) and *valid-until* (VU). VA precedes FU, which in turn precedes VU. A network status document is valid between VA and VU and can be used during this time period. A document is fresh until a new consensus becomes valid. These times overlap such that at any point in time at least three network status documents are valid. But only one document is fresh, which should be preferred over the other documents, if a client has more than one valid document.

Relays can be configured to act as directory caches. The caches download network status documents and relay descriptors directly from directory authorities, whereas clients download the documents from directory caches in order to better balance the load on directory authorities. All the communication, both upload and download, is performed via HTTP.

Relay descriptors and extra info documents

All documents produced as part of the directory protocol specified in [22] must be signed by the party producing it, otherwise these documents are invalid. Because of that, every document can be verifiable attributed to the document producer. All documents are ASCII text documents and consist of a variable number of items. Each item has a keyword and following the keyword specified attributes valid for this item, for example "**keyword** *attribute1 attribute2*". In general, items are terminated by a newline, such that every item is one line. But items can also be multiline, for instance document signatures. The exact format of all documents with their allowed items is specified in [22]. This paper focuses on the main points to keep the description precise and understandable.

Relay descriptors and extra info documents are generated and uploaded to the directory authorities by default every 18 hours or when the content changed significantly. The relay descriptor contains items such as (most important, no complete list):

- **router**: IP address and ports to connect to the relay.
- **fingerprint**: Hash value of the relay's identity key.
- **published**: Timestamp when the descriptor was generated.
- **bandwidth**: An estimated bandwidth the router is willing to contribute to the network.
- **uptime**: The time in seconds the Tor process is running.
- **hibernating**: Relay is in hibernating mode and should not be used for circuits. For example, the supplied bandwidth is exhausted.
- **onion-key** and **ntor-onion-key**: Keys used to establish session keys between clients and relays, see Subsection 2.3.3.
- **signing-key**: The relays long-term identity key.

- **accept** and **reject**: Specifies the allowed and disallowed exit connections. Used to advertise exit policies.
- **hidden-service-dir**: The relay serves descriptors for hidden services.
- **extra-info-digest**: Links to a corresponding extra info document.
- **router-signature**: Signature of the relay descriptor calculated with the identity key.

An example of a relay descriptor produced by the relay with the nickname `Atlantic` is shown below in Figure 2:

```
router Atlantic 85.25.43.84 443 0 80
platform Tor 0.2.4.23 on Linux
protocols Link 1 2 Circuit 1
published 2014-09-13 04:46:36
fingerprint 52FF FF0C 1BD5 58DC 50D4 8F31 B249 463C 3F42 08EC
uptime 2341302
bandwidth 26214400 32768000 27157506
extra-info-digest E9F7C5A850FE737F7B05FFD479239E705A43ADEF
onion-key
[...]
signing-key
[...]
hidden-service-dir
contact frserv@safe-mail.net - 1KUgpf8uXg5gQUSozb3FTSJo3M3YbfixYd
ntor-onion-key maPq13ndyx7CuoLm/D7fZs7TinYfb0crWWh2uHlcdzY=
[...]
accept *:79-81
[...]
reject *:*
router-signature
[...]
```

Figure 2: Relay Descriptor of `Atlantic`, Received 2014-09-13

Extra info documents are not necessary to create and contain mainly statistical data about the usage of a relay: how many data a relay read and wrote, how many requests from different IP addresses and different countries it received, how many data was transferred as a directory cache, how many data as part of normal Tor traffic and how many data as an exit node and so forth. These statistics are gathered, stored and visualised at Tor's Metrics Portal available at `https://metrics.torproject.org/`. For more information on how the data contained in extra info documents is used see [25].

From the received relay descriptors directory authorities generate *microdescriptors*. These microdescriptors contain only the minimal necessary information about relays clients need in order to choose relays for a circuit and to communicate with them, for instance **onion-key** and **ntor-**

onion-key, IP address and port as well as the exit policy. They are stripped down versions of relay descriptors to save bandwidth and time during download and directory authorities can directly transform relay descriptors to microdescriptors.

Key certificates

Key certificates have the following format as exemplary shown in Figure 3 for the directory authority `gabelmoo`. They contain a version number of the certificate format **dir-key-certificate-version**, a **fingerprint** as the hash of the authority identity key, an optional IP address plus port for the authority, a timestamp **dir-key-published** when the certificate was published, a timestamp **dir-key-expires** after which the authority signing key is invalid, the authority identity key **dir-identity-key** itself, the authority signing key **dir-signing-key**, a cross-certificate **dir-key-crosscert** and a signature of the key certificate **dir-key-certification**. The signature ensures that the authority signing key belongs to the correct directory authority. The cross-certificate is a signature with the authority signing key of the authority identity key. Thus, the two keys certify each other and both signatures must be valid to accept the key certificate. This proves that the directory authority is in the possession of both keys. Otherwise a directory authority could sign the public component of a authority signing key without possessing the private component and could generate a valid key certificate.

```
dir-key-certificate-version 3
fingerprint ED03BB616EB2F60BEC80151114BB25CEF515B226
dir-key-published 2014-04-07 23:01:16
dir-key-expires 2014-12-07 23:01:16
dir-identity-key
[...]
dir-signing-key
[...]
dir-key-crosscert
[...]
dir-key-certification
[...]
```

Figure 3: Key Certificate of `gabelmoo`, Received 2014-09-13

Vote and consensus documents

In order to generate a consensus, i.e. a network status document, the directory authorities perform the following five steps:

1. Create votes based on known relays.
2. Exchange votes with the other directory authorities.
3. Create a consensus document and sign it.
4. Exchange signatures of the consensus document with the other directory authorities.
5. Serve the consensus document to clients.

The vote is a document of one single directory authority, which contains the status of the Tor network as it is currently seen by this authority. This includes all the relays the authority knows about due to uploaded relay descriptors and the perceived status of the relays. In this process directory authorities assign flags to relays, which tell clients roles and properties of relays in order to make better decisions while constructing circuits. The most important flags are the following:

- *Authority*: The relay is a directory authority.
- *BadExit*: The relay is unusable as an exit node.
- *Exit*: The relay is suitable as an exit node.
- *Fast*: The relay can handle high-bandwidth circuits.
- *Guard*: The relay is suitable as an entry node.
- *HSDir*: The relay serves hidden service descriptors.
- *Running*: The relay is up and running.
- *Stable*: The relay can handle long-lived circuits.
- *V2Dir*: The relay acts as a directory cache.
- *Valid*: The relay does not run a broken Tor version and is not blacklisted as suspicious by the directory authority.

Under which circumstances flags are assigned to relays is specified in detail in [22]. In addition to assigning flags, a set of bandwidth scanners measure the bandwidth of each relay and feed that data back to the directory authorities, which utilise it to provide more accurate information about the relays' bandwidth [26]. Thus, directory authorities do not need to trust the reported bandwidth values of relays.

In the second step the directory authorities exchange the votes. In this step every authority learns, what the other authorities think about the current status of the network. If one authority does not know a relay another authority knows, the authority can download the relay descriptor from the other authority to learn the unknown relay. But this relay will only be part in the next vote. After obtaining all the votes from all directory authorities each authority independently creates a consensus document by combining the votes and signs it (step three). In general, when directory authorities have conflicting views of the network or about a specific relay, a majority vote is conducted to resolve that conflict.

It is important that all the directory authorities generate *exactly* the same consensus document. In order to accomplish this the Tor specification of directory authorities [22] defines precisely how to compute the consensus based on the votes. Multiple methods for this computation exists, which are called *consensus methods*, and in general newer methods extend the older methods. In the votes directory authorities include the consensus methods they support. The newest consen-

sus method supported by two-thirds of the directory authorities is used to calculate the consensus. The final consensus document is signed by all directory authorities.

In the fourth step the directory authorities exchange the signatures of the consensus document and all signatures are attached to the document. This is possible, because all directory authorities signed the exact same document. When clients download the consensus in step five, they can therefore verify that all directory authorities agreed upon the same network status.

```
network-status-version 3
vote-status consensus
consensus-method 17
valid-after 2014-09-13 08:00:00
fresh-until 2014-09-13 09:00:00
valid-until 2014-09-13 11:00:00
voting-delay 300 300
client-versions 0.2.3.24-rc,0.2.3.25,0.2.4.17-rc,0.2.4.18-rc, [...]
server-versions 0.2.4.23,0.2.5.6-alpha,0.2.5.7-rc
known-flags Authority BadExit Exit Fast Guard HSDir Running Stable V2Dir Valid
params CircuitPriorityHalflifeMsec=30000 NumDirectoryGuards=3 NumEntryGuards=1
  NumNTorsPerTAP=100 UseNTorHandshake=1 UseOptimisticData=1 bwauthpid=1
  cbttestfreq=1000 pb_disablepct=0 usecreatefast=0
[...]
dir-source gabelmoo ED03BB616EB2F60BEC80151114BB25CEF515B226
  212.112.245.170 212.112.245.170 80 443
contact 4096R/C5AA446D Sebastian Hahn <tor@sebastianhahn.net> -
  12NbRAjAG5U3LLWETSF7fSTcdaz32Mu5CN
vote-digest 3A9240CD6666793B6954C2E7D23F8523CEE85B9B
[...]
r Atlantic Uv//DBvVWNxQ1I8xsklGPD9CCOw jSegCkirIunDfYZSfOo+bv9brzk
  2014-09-13 04:46:36 85.25.43.84 443 80
s Exit Fast Guard HSDir Running Stable V2Dir Valid
v Tor 0.2.4.23
w Bandwidth=99900
p accept 20-23,43,53,79-81,88, [...]
[...]
directory-footer
bandwidth-weights Wbd=708 Wbe=0 Wbg=4034 [...]
[...]
directory-signature ED03BB616EB2F60BEC80151114BB25CEF515B226
  CD7159A8DE14BC6BDC7E5E1E51ADC89E162FCA08
[...]
```

Figure 4: Network Status Document, Received 2014-09-13

Vote and consensus documents contain different categories of information, see Figure 4 above as an example for a consensus document. This example includes entries for the directory authority `gabelmoo` and the relay `Atlantic`. The first category consists of information to handle the voting

and consensus building process. This includes inter alia the document type (vote or consensus), supported consensus methods, the VA, FU and VU times, information about the directory authorities and flag assignment parameters. These parameters, which are only contained in votes, specify thresholds an authority used to assign flags to relays.
The second category is information about the Tor network in general. This includes a list of recommend Tor software versions to warn users about outdated software, known flags and client parameters. The client parameters are used to modify the behaviour of the Tor software. This has the advantage that the behaviour of the clients can be changed based on the parameters without the need to distribute a new version of the software. For example, this can be utilised to migrate from an old protocol version to a new version. When the new behaviour is implemented, a new version of the software can be shipped with the new behaviour initially deactivated. After most users and relays upgraded to the new software version, a consensus parameter can be set to activate the new protocol. This allows a smoother transition to new versions of Tor, which requires users and relay operators to upgrade the software. In addition, thresholds and other constants can be tweaked in this way to experiment with other values without the need to redistribute Tor to a lot of people.
The third category of information included in these documents is information about each relay similar to the relay descriptors. It contains information about how to communicate with a relay, the relay's exit policy, the assigned flags and the bandwidth of relays, either the advertised, self-reported bandwidth from relays or the measured bandwidth from the bandwidth scanners.
In most cases both votes and consensus documents contain the same information, but some items are only included in one document, because it is not necessary to include them in the other. One important component of consensus documents are **bandwidth-weights**. These bandwidth weights are calculated by the directory authorities and used by clients in the path selection algorithm to choose relays for a circuit, see Subsection 2.3.2. The weights are used to optimise the performance and the load balancing of the Tor network.

In addition to the normal consensus, directory authorities also compute a *microdescriptor consensus*. The main difference is that the microdescriptor consensus does not contain exit policies of relays, because this information is already contained in the microdescriptors of relays. Furthermore, a link to the corresponding microdescriptor of the relay is included as well, such that a client can locate the corresponding microdescriptors. The goal is to avoid duplication of information in consensus documents and microdescriptors.

Using network status documents and relay descriptors

In regular intervals clients download network status documents to learn the status of the Tor network. Additionally, they download relay descriptors corresponding to the relays in the status document and keep these descriptors up-to-date. This ensures that clients always have updated information about the network and are able to build circuits and to use the network. All necessary information are contained in these documents. If clients download the microdescriptor consensus, they also download the corresponding microdescriptor relay descriptors. They use key certificates and the authority signing keys contained in them to verify that they received correct and untampered network status documents.

2.3.2 Path Selection

Once clients have a network status document and enough relay descriptors they can start to select nodes and to build circuits through the Tor network. But how do they decide what nodes they want to use for a particular circuit? This is specified in the "Tor Path Specification", see [27]. A *path* is an ordered sequence of three nodes in the network. The client connects to all three nodes on a path in order to establish a *circuit* through the Tor network, which is subsequently used to transmit data. The circuit is extended one hop at a time. A successfully built circuit is utilised to transport a *stream* of data. A stream is an end-to-end connection from the client to the destination with the last node on the path making the direct connection to the destination. Once a circuit construction finished, a stream can be attached to the circuit and a connection established with the destination. Finally, this stream transports all data exchanged between the client and the destination over the circuit.

Tor builds circuits pre-emptively based on the observed usage of Tor in the last hour. Because of that, a set of circuits is always available and can be used for streams, such that the user does not need to wait for the construction of a new circuit upon request. A circuit is called a *clean* circuit, if no user traffic was transported over this already completed circuit before. Furthermore, Tor builds circuits on demand, if no clean circuits can serve the user's request. In addition to circuits used for transportation of user data, Tor relays build circuits for internal reachability and bandwidth tests. For the reachability test the relay creates a circuit with itself as the last hop in order to determine that other relays and clients can connect to it. Only if the reachability test finished successfully, the relay publishes its relay descriptor. Additionally, relays create test circuits to measure their bandwidth more accurately and include this information in the relay descriptor.

The method of building circuits is called *telescoping*, because the circuit is only extended to the next hop, if the client could connect to the previous node without an error, similar to the extension of a telescope. This method can be utilised to "cannibalise" clean circuits for a purpose different than the originally intended purpose, for instance to replace the exit node on the circuit in order to allow the connection to another destination based on the exit policy. Or the circuit could be extended to another hop on demand. Furthermore, Tor keeps track of the time needed to create circuits. This is used to calculate a timeout after which Tor gives up to build a specific circuit, marks this circuit creation as failed and does not try to establish this circuit any longer.

Tor chooses nodes based on the flags assigned to them by the directory authorities, their bandwidth and some additional constraints. For a relay having a high bandwidth it is more likely to be picked as a node for a path. The goals of the path selection algorithm are circuit performance, load balancing and security. Clients should use circuits, which offer them a high performance, but at the same time the load needs to be distributed to all nodes in the network in order to avoid that a few fast nodes are overloaded and must handle most of the traffic. In addition, it should be difficult for an attacker to come into the position to control both the entry and exit nodes on a path, because this would allow effective traffic correlation attacks. The following rules apply during the selection of nodes for a single path with the exit node being chosen first:

- No relay is picked twice.
- Not more than one relay from the same *family* is picked. Relay operators are encouraged to specify during relay configuration, that relays controlled by the same operator belong to the same family. Thus, every relay on a path should be controlled by a different operator.
- Only pick one relay from the same /16 subnet. This ensures that relays are not closely located and makes it less likely that two relays can be observed by an attacker at the same time.
- Relays must have the *running* and *valid* flag in order to avoid turned off or malicious nodes.
- The first node must have the *guard* flag, the last node the *exit* flag.
- For long-lived circuits which are expected to be open for a long time period, for example remote logins with SSH, only nodes with the *stable* flag are picked. This makes it less likely that the circuit breaks during operation because of unreliable nodes.
- Exit nodes are picked according to their exit policy. This ensures up front that the connection to the destination will not be refused by the exit node due to a violation of the exit policy.
- Node selection is weighted by the node's bandwidth. Fast nodes offering high bandwidth are preferred.
- The **bandwidth-weights** from the consensus document determine the probability of selecting nodes for a particular position on the path, for instance the probability to pick a node with the *guard* flag as an entry, middle or exit node. For example, nodes with the *guard* flag are preferred as the first node and nodes with the *exit* flag as the last node. The weighting depends on the bandwidth all guard respectively exit nodes contribute to the network compared to the overall bandwidth of all relays in the network. This helps to load balance the traffic in the network.

The concept of *guard nodes*, i.e. relays with the *guard* flag, is important for the security of Tor. Recall that in many cases the goal of an attacker is to control both the entry and the exit node on a circuit in order to correlate traffic. In general, if an attacker controls C out of N relays and nodes are selected at random, the probability of choosing a compromised path with the attacker controlling both entry and exit node is $\left(\frac{C}{N}\right)^2$. Over time this probability goes to 1, when a client keeps building circuits at random. The attacker only needs to wait until the client creates a compromised circuit. The concept of guard nodes try to mitigate this attack [27].

Tor chooses a small set of guard nodes, currently three, and always uses one of the guard nodes as the entry node. When selecting guard nodes, there are two possible outcomes. First, the guard is controlled by the attacker and all circuits can be compromised, or second, the guard is not controlled by the attacker and the attacker never has a chance to compromise a circuit. If the number of attacker controlled guards compared to all relays in the network is small, then the probability of picking a bad node is small as well. This increases the costs for an attacker, because he needs to run more nodes in order to increase the likelihood of a successful attack.

But this method also has a drawback. Because clients do not change their guards and new clients choose the same guard nodes, these nodes accumulate more and more clients and traffic they need to process. For this reason clients rotate their guard nodes currently every 8 to 12 weeks [23]. This helps to distribute the traffic over all guard nodes. This is always a trade-off between performance, load balancing and security and is subject to current research in order to choose the best parameters for this trade-off [28]. The Tor developers discuss to move to a single guard node and a rotation period of 9 to 10 months [29]. The goal is to further slow down attacks, because with these parameters attackers need to wait even longer before clients pick their controlled nodes. It is important to note that the design of guard nodes does not prevent traffic correlation attacks itself, instead it is designed to slow down attacks and to make attacks unattractive due to the required time and/or costs for running nodes.

2.3.3 Communication Protocol

Tor's communication protocol as specified in [30] defines all communications within the Tor network: relay-to-relay communication, client-to-relay communication and the end-to-end communication between the client and the destination outside the Tor network. This protocol supplies the functionality of anonymous communication. Tor uses various cryptographic algorithms as building blocks of the protocol: AES in the counter mode with a 128 bit key as a symmetric cipher, RSA with a 1024 bit key as a public-key cipher and SHA-1 as a cryptographic hash function. All relays in the network own three different public-private keypairs:

- A long-term *identity key* used as the relay's identity and to sign certificates as well as relay descriptors and extra info documents. Clients learn this key from relay descriptors.
- A medium-term *onion key* used to establish an authenticated and encrypted connection between a client and a relay on a circuit. This key is also part of the relay descriptor.
- A short-term *connection key* used to negotiate TLS link connections between two adjacent relays and between a client and an adjacent relay. This key is rotated at least once a day.

Establish TLS connections

Direct link connections between two adjacent relays respectively a client and a relay is secured with TLS. The client establishes a TLS connection with the entry node and every node on the path does the same with its successor. These TLS connections are used to transport the Tor protocol, for instance creating circuits and transferring data over streams. Multiple circuits can share a single TLS connection. The goal is to establish an encrypted connection and to mutual authenticate both ends of the connection. It is important that a client or relay knows it is talking to the correct relay and not to an imposter, who performs a Man-in-the-Middle attack. The utilisation of TLS also prevents that an outside attacker can modify the data sent over the connection.

There exists three specified versions of the handshake to establish a TLS connection: *certificates-up-front*, *renegotiation* and *in-protocol*. The first two handshakes authenticate both parties with standard TLS methods, whereas authentication with the third handshake is performed with a

Tor-specific protocol. Especially the third handshake is designed in a way that it looks to an observer like an ordinary TLS connection between a Web server and a browser. The goal is to make it difficult to determine that a TLS connection is a Tor connection by just looking at the TLS traffic. For the same reason the encryption algorithms advertised in the TLS handshake as supported by Tor mimics the cipher list of popular Web browsers. Furthermore, if clients use TLS certificates during the handshake, these certificates are constructed in a way that they do not leak identifying information or identify Tor traffic. In addition, new certificates are created in the case the IP address of a user has changed. See the specification [30] about how clients and relays know when to use which handshake.

With the certificates-up-front handshake the initiator of the connection sends two X.509 certificates to the responder during the TLS handshake. The first certificate contains the short-term connection key. This certificate is signed with the long-term identity key, which is part of the second, self-signed certificate. The responder answers the request with a similar two-certificates chain in order to finish the handshake. Because both initiator and responder know the identity keys from the relay descriptors, they can verify that they are actually connected to the right relay. At this point both parties have successfully authenticated each other and established an encrypted TLS connection usable for further communication.

Using the renegotiation handshake the initiator does not send any certificate during the initial TLS handshake and the responder answers only with a single connection certificate. This behaviour is similar to a browser establishing a TLS connection with a Web server, where only the Web server sends a certificate to the browser to authenticate itself. After this initial TLS handshake is finished, the initiator starts a TLS renegotiation with both initiator and responder sending two certificates in the same manner as with the certificates-up-front handshake.
The in-protocol handshake has the same first step as the renegotiation handshake. But the second step uses the Tor protocol for authentication. This is explained in more detail in a following subsection.

Cell format

After a TLS connection is successfully established, this connection is used for all further communications. For this purpose Tor uses fixed-size *cells* with two basic types of cells: *control cells* and *relay cells*. Control cells are exchanged between two adjacent nodes and are always interpreted by the node receiving them, for example the creation of circuits uses control cells. In contrast, relay cells are end-to-end communication between the client and the last node on a circuit and are utilised for stream management and end-to-end transportation of data. Relay cells are just forwarded by the intermediary nodes on the circuit. In addition to the fixed-size cells Tor has cells with a variable length.

A cell has the following format, see Figure 5 on the next page: a four byte circuit identifier *CircID* followed by an one byte *Command* followed by 509 bytes of payload, in total 514 bytes[1].

[1] Until recently Tor used a CircID of two bytes. Thus, a cell was 512 bytes long.

The randomly chosen CircID associates a cell with a particular circuit and is different for every TLS connection. Data travelling over a circuit has a different circuit identifier on each link between two nodes. The command describes how a relay processes the cell and how the payload is formatted. Payload shorter than 509 bytes is padded with zeros. A variable-length cell has in addition to *CircID* and *Command* a two bytes *Length* field after the command, which specifies the length of the following payload.

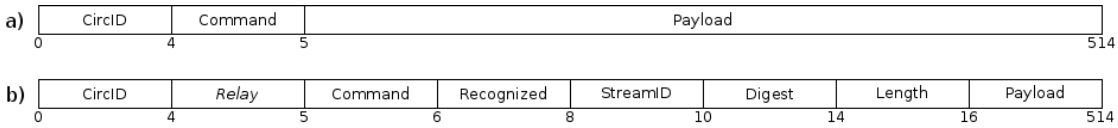

Figure 5: Format of Control a) and Relay b) Cells

Relay cells have the cell command `relay` and add an own header to the start of the cell's payload as illustrated in the figure above. This header consists of an one byte *Command* to identify the relay cell, a two byte *Recognized* field set to zero, a two byte identifier *StreamID* of the stream the relay cell is associated with, a four byte checksum *Digest*, the *Length* of the remaining payload and the actual relay cell payload. The relay cell header and the payload is AES-encrypted.

Initialise connections

In the case of the renegotiation and in-protocol handshake the connection between two Tor instances, either two relays or a client and a relay, must be initialised, before circuits can be built and data transmitted over the connection. Especially, they must agree on a version of the protocol they will use. Currently Tor has four versions:

1. The first version uses the certificate-up-front handshake.
2. The second uses the renegotiation handshake and introduced variable-length cells.
3. The third uses the in-protocol handshake.
4. The fourth and most current version implemented circuit identifiers of four bytes.

With both renegotiation and in-protocol handshake the first cell both Tor instances send is a `version` cell, i.e. the command in the cell header corresponds to `version`. This cell contains the numbers of the link protocol versions supported. The protocol version with the highest number both parties understand is selected for further communication. Following that they immediately exchange `netinfo` cells, if the renegotiation handshake is used. These cells contain a current timestamp, the IP address of the other party and the own IP addresses. IP addresses are encoded in the form Type-Length-Value (TLV). The type either specifies a hostname, an IPv4 address or an IPv6 address, length the number of bytes of the address and value is the actual address. On receiving the `netinfo` cell Tor checks, whether it is connected to the correct IP address of the other Tor instance. Otherwise an attacker could mount a Man-in-the-Middle attack by tricking a relay to connect to an attacker controlled relay, although he would not be able to read the encrypted traffic.

For the renegotiation handshake this process is enough to set up the connection. With the in-protocol handshake both parties still need to authenticate each other. This authentication process follows these steps:

1. The initiator sends a `version` cell.
2. The responder answers with a `version`, a `certs`, an `auth_challenge` and a `netinfo` cell. The `certs` cell includes certificates similar to the certificate chain used with certificate-up-front and renegotiation to identify and authenticate the responder.
3. The initiator responds with a `certs`, an `authenticate` and a `netinfo` cell to authenticate himself. If he does not want to authenticate, he just sends a `netinfo` cell. The `authenticate` cell must include the challenge from the previously received `auth_challenge` cell.

Both `certs` cells include two X.509 certificates. The responder sends exactly the same two-certificates chain as he would do with the certificates-up-front and the renegotiation handshake, i.e. the TLS link certificate with the short-term connection key used for the TLS connection and an identity certificate with the long-term identity key signing both certificates, which is enough for the initiator to authenticate the responder. The `certs` cell from the initiator to the responder (step 3) includes an identity certificate as well and an authenticate certificate, which consists of an *authentication key*. Again, both certificates must be signed by the initiator's identity key. Checking these certificates is not sufficient to authenticate the initiator by the responder. He must verify the `authentication` cell as well.

The `auth_challenge` cell sends a randomly chosen 32 bytes challenge to the initiator, who must include this challenge in his `authenticate` cell answer. This cell consists of SHA-256 hashes of the initiator's identity key, the responder's identity key, all the bytes sent so far from the responder to the initiator including the `auth_challenge` cell, all the bytes sent so far from the initiator to the responder, the responder's TLS link certificate as well as a secret value based on TLS parameters including the TLS secret master key, a 24 bytes random value and a signature of the SHA-256 hash value of all the previous fields using the authentication key included in the `certs` cell for signing. The responder can check all these fields and can verify the signature in order to authenticate the initiator, because the initiator is the only entity able to create the `authenticate` cell and to sign it correctly with the private part of the authenticate key. This process is necessary, because in contrast to the other two handshake methods the initiator does not use a TLS link certificate, which can be used for authentication. This finishes the initialisation of connections with the in-protocol handshake.

Create circuits

Circuits are created one hop after another. Tor clients send a `create` cell to the entry node including the first half of a handshake to authenticate the node. The node replies with a `created` cell and the second half of the handshake. In order to extend the circuit the client sends an `extend` relay cell to the entry node, which unwraps the cell, constructs a create cell with the payload of the `extend` relay cell and forwards the `create` cell to the middle node. If the entry node does not have a TLS connection to the middle node, this connection is established first as previously

described. The middle node answers with a `created` cell to the entry node, which wraps the answer into an `extended` relay cell and transmits this cell to the client. The same process is repeated in order to extend the circuit to the exit node with the client sending the `extend` relay cell to the middle node. In the same manner the circuit could be extended to even more hops.

Tor supports two authentication handshakes during the exchange of `create` and `created` cells: the original *TAP* handshake, which is based on the classical Diffie-Hellman (DH) key exchange [31], and the newer *ntor* handshake as proposed in [32]. TAP is slow due to the number of operations necessary to compute the DH key exchange [13]. With ntor the original DH key exchange is replaced with a version based on elliptic curves, which is able to speed up the handshake significantly [33]. But why is another authentication performed in addition to the link authentication? The client has only an authenticated TLS connection with the entry node, but the middle and the exit node must also be authenticated by the client, such that the client can be sure to use only the nodes as chosen during path selection. Furthermore, both TAP and ntor are utilised to negotiate session keys between the client and each node on the circuit.

With the introduction of the ntor handshake the Tor developers had to change the protocol, because `create` / `created` and `extend` / `extended` assume only the TAP handshake. In order to support ntor and to make the protocol more flexible `create2` / `created2` and `extend2` / `extended2` cells are defined. The payload of `create2` has the form of TLV: type of handshake, length of the handshake data and the handshake data itself. The two types currently specified are TAP and ntor. The `created2` cell consists of a length field and the reply handshake data. The old `create` / `created` cells contain only the handshake data as defined for TAP.

The `extend2` and `extended2` relay cells have the same handshake payload as `create2` and `created2`, respectively. In addition, `extend2` relay cells define a variable number of *link specifiers*. These link specifiers, again in the form TLV, are either an IPv4 or IPv6 address plus port or an identity fingerprint of the next hop on the circuit. The fingerprint consists of the SHA-1 hash value of the relay's identity key. The link identifiers are required to tell a relay the next node to whom the circuit should be extended. The `extend` relay cell only contains an IPv4 address, a port, the TAP handshake data and the relay's fingerprint. The payload of the `extended` relay cell is the same as of the `created` cell. A node receiving an `extend` or `extend2` relay cell copies the embedded handshake data, creates a `create` respectively `create2` cell with this data and sends the cell to the node as identified by the link specifier. In the same way the response handshake data is copied and sent to the client in `extended` / `extended2` relay cells.

TAP handshake

The TAP handshake uses the Diffie-Hellman key exchange. The client chooses a value x and computes $g^x \bmod p$ with g and p given and encrypts the result with the relay's RSA public onion key learned from the relay descriptor. The encrypted value is sent to the relay as the handshake payload of a `create` or `extend` cell. The relay chooses a value y and calculates $g^y \bmod p$. The result is sent back to the client together with a number KH in a `created` or `extended` cell. Both parties can then calculate a shared secret $g^{xy} \bmod p$. This shared secret is used to derive

session keys, inter alia the number KH. Because the first half of the handshake is encrypted with the relay's public onion key, only the right relay can decrypt it and derive the session keys. The number KH included in the response demonstrates that the relay was able to calculate the session keys and authenticates the relay to the client.
In order to derive session keys the number i is concatenated to the shared secret and the result hashed. This procedure is repeated with initially $i = 0$ and i incremented by 1 in every step. All hash values are concatenated and five keys are taken from this: the response number KH, the forward digest Df, the backward digest Db, the forward encryption key Kf and the backward encryption key Kb. The usage of these keys is explained in a following subsection.

ntor handshake

The ntor handshake is built upon the elliptic curve function *Curve25519* as specified in [33]. A relay has a ntor onion keypair (b, B) based on this curve with b being the private key and B the public key. B is published in the relay descriptor as the ntor onion key. The client generates a temporary keypair (x, X) and sends the fingerprint of the relay, the identifier of B and X to the relay as the first half of the handshake. The relay generates a temporary keypair (y, Y), computes an authentication value based on X, y and b and replies with the authentication value and Y as the second half of the handshake. Only the relay knowing the private keys y and b can generate this value. The client verifies the authentication value with x, Y and B. Furthermore, both compute a shared secret used for session key derivation.
The key derivation function (KDF) of ntor is a little bit different than the KDF of TAP. The TAP KDF always uses the same shared secret during concatenation with i. In contrast, ntor's KDF concatenates i to the hash value from the previous step with the shared secret only used as the initial value. But again the same five keys KH, Df, Db, Kf and Kb are generated.

Close circuits

Circuits are closed either on request, for example if all streams on a circuit finished, or in the case of an unrecoverable error. Circuits can be closed completely or hop-by-hop. When a client or relay want to close a circuit, they send a `destroy` cell to the adjacent node, for instance to the entry node in the case of a client. On receiving the `destroy` cell the relay frees all resources associated with the circuit, including any connections to servers outside the Tor network, if the relay is an exit node. If the relay is not the last node, the `destroy` cell is forward to the next node, propagating the closing of the circuit.

Circuits can be closed hop-by-hop similar to the creation of circuits by sending a `truncate` relay cell to a particular relay. The receiving relay sends a `destroy` cell along the circuit and replies to the initiator with a `truncated` relay cell to signal that the closing request was processed. This procedure can be used, for example, to remove the exit node from a circuit and to extend the circuit again to another exit node.

If a circuit breaks due to an unrecoverable error between two adjacent relays, the relay closer to the client sends a `truncated` relay cell to the client and the other relay sends a `destroy` cell in the direction of the circuit end. Because of that, all nodes the client cannot reach anymore are

closed down cleanly and the remaining circuit is kept working. This feature is useful in order to recover from errors, because circuits can be rebuilt starting with the last reachable node. Both `truncated` and `destroy` cells contain a single byte value as the payload indicating the reason for the closing. The `truncate` relay cell does not have any payload.

Processing relay cells

After the client created a circuit, authenticated all relays and established session keys, the circuit can be used for sending data. The end-to-end communication between the client and the exit node is handled by relay cells. Sending cells from the client to the exit node is called *forward* direction, from the exit node to the client *backward* direction. The keys Kf and Df are used in the forward direction, Kb and Db in the backward direction. A client encrypts the cell's payload with all three Kf keys using AES and transmits the cell along the circuit. Every node on the circuit removes one layer of encryption by decrypting the cell with its own Kf, thus, the exit node receives the original plaintext. On a reply back to the client all nodes add one layer of encryption by encrypting the cells payload with Kb. For this reason only the client can read the payload, because only he possesses all Kb keys.

In order to preserve the integrity of the cell a checksum is included in each relay cell, i.e. the *Digest* field of the relay cell header. For each relay on the circuit Tor keeps a running digest and includes the first four bytes of this digest as the checksum. In the forward direction the running digest is the hash value of all cells sent from the client to a particular relay, in the backward direction all cells sent from the relay to the client. The running digest is initialised by Df respectively Db and only includes cells, which are destined for a relay, i.e. no cells just forwarded by a relay. This ensures that the receiver of a cell would detect any manipulations, because he can compare the running digest with the expected digest in the relay cell's header.

When a relay receives a relay cell in the forward direction, it decrypts the payload and checks whether the *Recognized* header field is set to zero. Recall that this field is always initialised by the sender as zero. If the field is not zero, the relay just forwards the decrypted cell, because it is not the intended recipient as it cannot decrypt the plaintext correctly. If the field is indeed zero, the relay checks the digest and in the case of a correct match processes the cell. If the digest does not match, the cell is forwarded as well. This process guarantees that a relay can easily identify, if it is the intended recipient of a relay cell. It acts as an address for the cell's destination.

It should be noted that with the techniques described in this subsection relay cells can be sent to any relay on a circuit, not just to the exit node by "addressing" the right relay and only adding encryption layers for the relays before the recipient.

Manage streams and transfer data

When an application wants to send data over the Tor network, it connects to the Tor client via Tor's SOCKS interface and request a connection to the remote host. Tor chooses an open circuit, which is able to handle the request, or creates a new circuit if necessary. Then it sends a `begin` relay cell to the exit node to open a stream. This cell contains the hostname, IPv4 address or

IPv6 address and the requested port of the remote host as supplied by the application through the SOCKS interface. On receiving the `begin` relay cell the exit node resolves the hostname if necessary and opens a TCP connection to the remote host on the specified port. The exit node replies with a `connected` relay cell, which includes the resolved IP address and a time-to-live (TTL) value. The TTL gives the number of seconds the resolved IP address should be cached. If the exit node cannot successfully connect to the remote host, it replies with an `end` relay cell.

After a stream is successfully established, Tor acknowledges the successful connection to the application via SOCKS. Afterwards, all TCP data handed from the application to Tor is packed in `data` relay cells and transmitted to the exit node. There the data in unwrapped and sent to the remote host. All data received from the remote host by the exit node is wrapped in `data` relay cells and transported back to the client, which hands the data to the application. If a connection is closed by either the remote host, the application or due to an error, the stream is closed by sending an `end` relay cell along the circuit, either in forward or backward direction. This cell contains a single byte with the reason for the closing. On receiving an `end` relay cell all connections associated with the stream are closed and no further data transmitted. The circuit itself is not closed, because different streams can share the same circuit.

If a relay serves directory information as described in Subsection 2.3.1, a client can fetch this information anonymously from the relay by sending a `begin_dir` relay cell to it. The relay connects to itself on the port used for serving directory information and establishes a stream as it would do after receiving a `begin` relay cell. The ordinary directory protocol is subsequently tunnelled over this stream. Furthermore, hostnames can be resolved anonymously as well. For this purpose clients send a `resolve` relay cell to the exit node containing the hostname to resolve. The exit node looks up the IP address belonging to the hostname and answers with a `resolved` relay cell, which contains a variable number of IP addresses of the form TLV plus TTL. This supports either IPv4 or IPv6 addresses as well as a reverse lookup, i.e. finding the hostname corresponding to an IP address. In addition, Tor implements a form of flow control for circuits and streams in order to avoid that they become congested. This is implemented via `sendme` relay cells, which signal to one end of the connection that a relay is able to receive more `data` relay cells. Otherwise transmission of cells is stopped after a fixed number of sent cells. For more details about flow control see [1] and [30].

Summary

In summary, clients and relays establish TLS connections to adjacent relays. The Tor protocol is transported over those connections. A client creates a circuit, authenticates all three relays on the circuit and negotiates session keys with them. Streams are opened on circuits in order to transport end-to-end data from the application to the remote host via relay cells, which are encrypted with the session keys. Figure 6 on the next page illustrates this process by showing what communication takes place between the different entities involved and what cells are transmitted. The example uses a two-hop circuit in order to provide a clear presentation. But this example could easily be extended to a three-hop circuit. This gives a good summary of the Tor protocol.

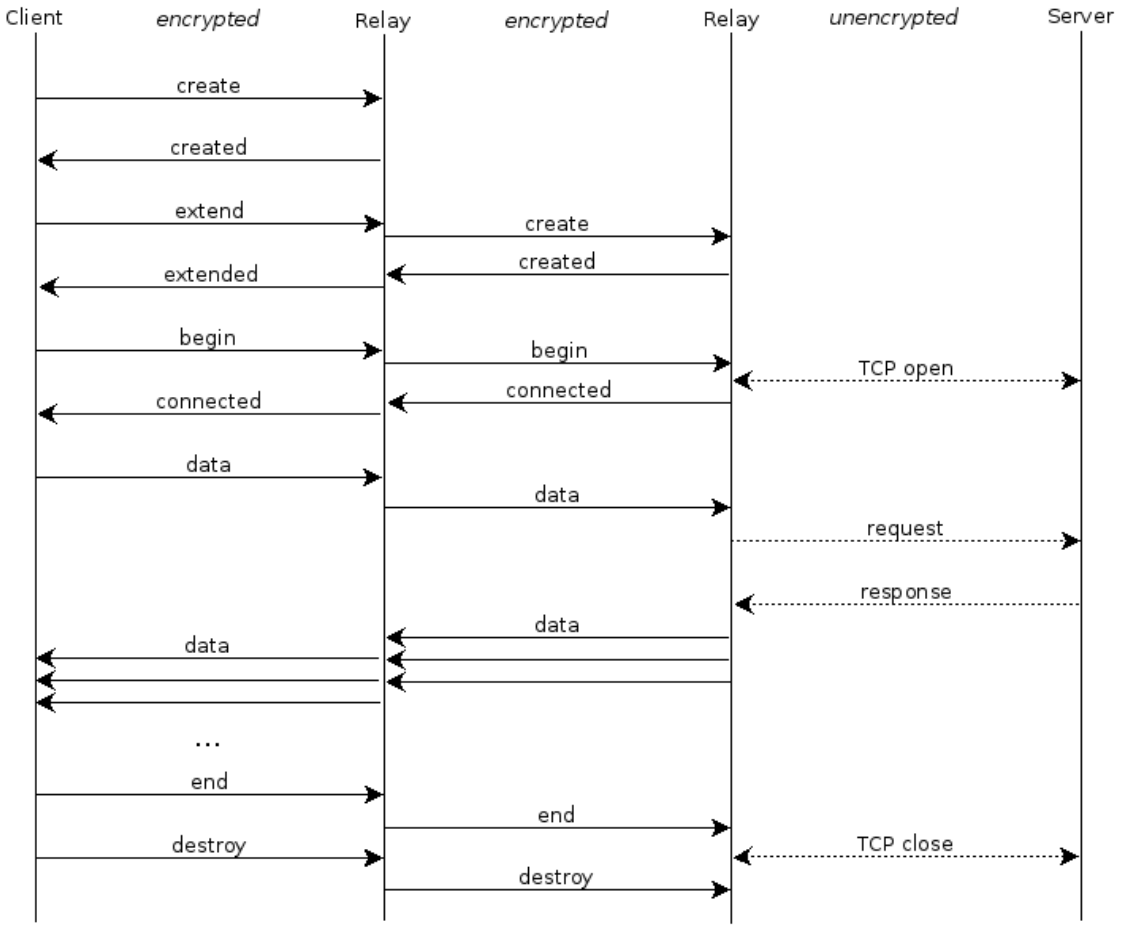

Figure 6: Retrieving Data from a Remote Host via a Two-Hop Circuit

2.4 Limitations and Pitfalls

Tor is used by millions of people everyday, who want to protect their anonymity. The previous sections explain in detail how Tor is designed in order to achieve that goal. But Tor has also some limitations and pitfalls users must be aware of. Tor anonymises any application using TCP instead of being tied to a single application, for example just Web browsing. But Tor only supports TCP and not UDP. Another approach would be to anonymise all IP packets regardless of the transport layer protocol used. But this has the great disadvantage that it would require in most cases modifications to the operating system's network stack. This would make the implementation more complex and less portable. Thus, Tor follows a middle approach in order to have a simpler design, which is portable to a lot of systems plus application-independent. This also helps to attract more users, because it is easier for them to install and use Tor for different purposes.

But Tor only anonymises the IP address of a user, it does not perform any protocol normalisation. If the protocol transported over Tor does not protect the user's anonymity, for instance it

contains the real IP address of the user as in the case of Bittorrent [34], Tor cannot help. Protocols must be designed in a way that they do not hurt the user's anonymity. The same holds for applications, which must be configured to not leak identifying information if combined with Tor. Another solution would be to layer a proxy between Tor and the application, which removes all identifying information from the traffic given to Tor. In addition, if no end-to-end encrypted protocol is used, the exit node can view the plaintext traffic leaving the network. For example, viewing a Web site with plain HTTP the exit node can see the viewed Web site, because the traffic leaving the exit node is unencrypted. Furthermore, multiple streams share a single circuit in order to avoid the overhead of constructing an own circuit for each stream. If two streams share a circuit and one streams leaks the user's identity due to a careless protocol, the other stream is deanonymised as well [34].
In general, Tor itself does not attempt to solve these application-level problems. Tor relies on protocol and application developers to solve it. For Web browsing the Tor Project offers the Tor Browser [4], which mitigates many problems regarding anonymity and Web browsing and gives users a tool to anonymously surf the Web. This is the biggest use case for Tor and requires special attention, because Web browsing and anonymity is itself a very difficult problem.

Furthermore, users must be aware of DNS leaks. When connecting to Tor via SOCKS, the application can provide either a hostname or an IP address to Tor. In the first case the exit node resolves the hostname before establishing the connection to the remote host. In the second case it uses the supplied IP address. This means that the application resolved the hostname itself. If this is performed outside Tor, the user leaks the destination to any entity observing the user's traffic. Because of that, users must ensure that applications transport both DNS resolves and TCP traffic over Tor.

By design Tor does not protect against traffic correlation attacks. But in fact the Tor protocol already specifies `padding` and `vpadding` control cells as well as `drop` relay cells, which can be used to create dummy traffic and could make traffic correlation attacks harder or impossible. But this would add additional load onto the Tor network. Until this security and performance tradeoff is better understood, mechanisms against traffic correlation attacks are not deployed [30].

Additionally, some parts of the Tor protocol show its age. Especially the cryptography primitives are partly outdated, for instance RSA with 1024 bit keys is too short. That was enough at the time Tor was originally designed, but not today with the massive increase of computing power over the last decade. The Tor developers are working on improving the situation [35], specifically by migrating identity keys to an elliptic curve public-key cipher [36]. But this needs time. Many parts of Tor must be changed and relays and clients migrated to the new software.

Tor is a great tool to provide anonymous communications over the Internet, but cannot solve the problem alone. Applications and protocols must be built with anonymity requirements in mind as well. Furthermore, user must be aware of Tor's limitations and pitfalls in order to not hurt their anonymity by using Tor in a dangerous way.

3 Tor Hidden Services

The Tor protocol as described in Chapter 2 provides anonymity to users, who want to access a public service anonymously. This allows users to view any public service. The service does not need to know about Tor. From the service's point of view the exit node is the user and makes the connection to it. Tor is mostly used in this way. But service operators may also have the desire to stay anonymous in order to provide the service without being identified as the service operator. For example, activists often want to publish sensitive information anonymously, because they fear repression from governments or corporations if they can be identified as the publisher. This is very important, for instance for whistle-blowers who want to uncover wrongdoing.

Tor's *location-hidden services* allow exactly that. Location-hidden services, sometimes also called *responder anonymity*, provides anonymity to a service and its service operator, where the real IP address of a hidden service is never revealed to anyone accessing the service. Thus, it cannot be determined, where the server running the service is physically located and who operates it. This makes it very difficult to shut down the service by governments or corporations and to censor the published information. In addition, this offers a protection against Denial-of-Service (DoS) attacks. An attacker cannot attack a hidden service directly, because he does not know its IP address. Instead he must attack through the Tor network, which can cooperatively defend the Denial-of-Service attack.

Hidden services are operated as part of the Tor network and users access them using the ordinary Tor client, ensuring anonymity for both users and service operators. This provides end-to-end encryption between the client and the hidden service, thus, nobody can listen on the communication between the user and the hidden service. Furthermore, the hidden service is authenticated to the user in a way that the user knows he is talking to the legitimate hidden service. Any TCP application can be configured as a hidden service. The Tor software acts as a proxy for the application and delegates all communication between the user and the application. Because of that, applications do not need modifications for being operated as a hidden service.

The classical example of a hidden service is a Web server for publishing information. But the hidden service design allows other interesting applications as well. For instance, TorChat[1] is a peer-to-peer instant messenger with a completely decentralised design employing hidden services. In fact, every user runs its own hidden service as part of the TorChat program. TorChat connects users to each other via the hidden service mechanism and all messages exchanged between two users are end-to-end encrypted with no intermediary able to listen on the conversation. Both parties stay completely anonymous. An attacker cannot find the location of users,

[1] Available at `https://github.com/prof7bit/TorChat`.

who is communicating with whom or what they say. TorChat is a great example of an application making use of the capabilities of Tor hidden services.

The following Section 3.1 introduces the general mode of operation of location-hidden services and describes the goals the Tor developers had in mind while designing them. Section 3.2 explains the current design and specification of hidden services, which is followed by an analysis of its limitations in Section 3.3. Based on this, Section 3.4 explains the proposal of a renewed hidden service design, which aims to fix these limitations. Section 3.5 concludes this chapter with a list of open research questions the new design does not address yet.

3.1 Introduction

The Tor developers had four design goals in mind when they created hidden services: *access control*, *robustness*, *smear-resistance* and *application-transparency* [1]. A hidden service can use access control mechanisms in order to determine, who is allowed to access the service. Furthermore, attackers should not be able to perform DoS attacks against the hidden service by just making many connections to the service. Robustness means that there is no single relay, which is solely responsible for handling one hidden service inside the Tor network. The hidden service should be able to operate even if relays taking part in the hidden service protocol shut down. Because of that, an attacker cannot render the service inoperable by attacking only a few relays. Smear-resistance in contrast should protect Tor relays, which handle hidden service traffic. An attacker should not be able to frame a relay and claim that this relay is offering an illegal or questionable hidden service. A relay should not be attributed as responsible for one hidden service. Application-transparency is the property that hidden services can be offered for any application without the need to modify it.

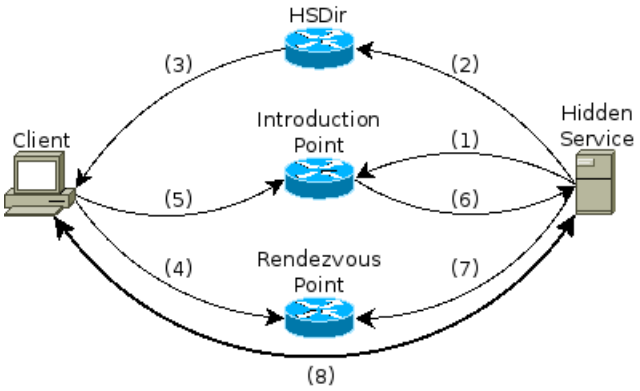

Figure 7: Overview of Hidden Services

The fundamental design of Tor hidden services is illustrated in Figure 7 above. The Tor software running on the hidden service computer is configured in a way that all connections to it are handed to the application offering the service. Tor does this transparently to the application. Every hidden service has a long-term identity key, which uniquely identifies the service. The public component is used as the hostname for the hidden service in form of a `.onion` address. This

virtual top level domain is interpreted as a request to access a hidden service, when submitted to Tor. The hostname has the form `z.onion` with `z` being the public key of the hidden service. In order to access a hidden service the following steps are performed (the enumeration corresponds to the numbers in Figure 7):

1. The hidden service chooses a set of ordinary Tor relays as medium-term *introduction points*, builds circuits to them and instructs them to act as its introduction points.
2. The hidden service creates a *hidden service descriptor*, which includes the service's public key and the selected introduction points, and publishes this descriptor to a set of *hidden service directories*. Every relay with the *HSDir* flag can act as a hidden service directory.
3. A user already knows the `.onion` address of the hidden service through a second channel. The client uses this address to download the hidden service descriptor from a hidden service directory in order to learn the introduction points.
4. The client picks a relay as a short-term *rendezvous point* and creates a circuit to it.
5. The client sends the rendezvous point together with the first half of an authentication handshake to one of the introduction points via a circuit and requests it to make a connection to the hidden service.
6. The introduction point forwards the connection request to the hidden service, which consists of the rendezvous point and the handshake data.
7. If the hidden service decides to communicate with the client, it establishes a circuit to the rendezvous point and sends the second half of the authentication handshake. The rendezvous point connects the client circuit and the hidden service circuit together in order to build a connection between the client and the hidden service. The authentication handshake is forwarded to the client.
8. The client authenticates the hidden service and both compute session keys based on the handshake data. This establishes an end-to-end encrypted connection between the client and the hidden service. Subsequently all communication is tunnelled over this connection via the rendezvous point.

This protocol secures the anonymity of both user and hidden service, because neither the hidden service directories, the introduction points nor the rendezvous point learn the real IP addresses of the user or the hidden service due to the fact that all connections are made through three-hop circuits. In fact, the introduction points do not even learn for which hidden service they are acting as introduction points [37]. If an introduction point is unavailable, the hidden service can pick a new one and include it in the hidden service descriptor. This makes the design robust, because if an attacker shuts down all introduction points, the hidden service just advertises new introduction points and is able to accept connections again. Smear-resistance is provided by using a different client selected rendezvous point for every connection. Thus, an attacker cannot just run a disreputable hidden service and frame a rendezvous point for sending illegal content, because he has no control over the choice of the rendezvous point. He can also not frame an introduction point, since those never transport any application data.

3.2 Current Hidden Services

The current protocol of Tor hidden service is specified in [37]. The Tor software runs on the same machine as the application, for instance a Web server, which should be made available as a hidden service. Tor is configured to forward all application data received via the hidden service protocol to a local IP address and port, where the application processes this data. Because of that, an application is not aware of the fact that it is offered as a hidden service. Users use hidden services in the same way they use non-hidden services via Tor. From the user's perspective there is no difference between a hidden service and a non-hidden service, except the `.onion` address.

3.2.1 Hidden Service Protocol

First the hidden service protocol for the communication between a client and a Tor instance running as a hidden service is described in this subsection with the explanation following the process illustrated in Figure 7 and explained in the previous section. Subsection 3.2.2 examines two different mechanisms hidden services can deploy in order to perform access control, i.e. to enforce that only authorised users can access the service.

Establish introduction points

Initially, the Tor instance running as a hidden service generates for every introduction point a short-term *service key* independent of its long-term identity key. Next it chooses a small set of three to ten relays as introduction points. This number of introduction points depends on the self-estimated popularity of the hidden service [38]. If the hidden service is more popular, more introduction points are selected. A circuit to each introduction point is created and an `establish_intro` relay cell sent to each of them. This relay cell contains the public service key component, a hash value based on the number KH from the authentication handshake between the hidden service and the introduction point and a signature of this information computed with the private service key component. The hash value is included in order to prevent replay attacks, since the value depends on the newly generated KH only the hidden service and the introduction point know. The introduction point verifies the signature and the hash value of the received relay cell and answers with an empty `intro_established` relay cell in the case of a positive outcome. The introduction point utilises the service key to associate requests from clients to the hidden service. The service key is used for this purpose instead of the hidden service identity key, because the introduction point should not know for which hidden service it serves as an introduction point.

Publish hidden service descriptor

After a hidden service has established its introduction points, it generates a hidden service descriptor. This descriptor has the same basic format as all directory documents described in Subsection 2.3.1. Every descriptor has a **rendezvous-service-descriptor** item, which contains a *descriptor-id*. This descriptor-id uniquely identifies the service descriptor and is used to publish it to different hidden service directories. It is calculated based on a hash value of the combination of the hidden service identity key, a current timestamp, a *descriptor-cookie* and a replica number. The timestamp changes every 24 hours and guarantees that the service descriptor is published to a different set of hidden service directories every 24 hours. The replica number is used to gener-

ate multiple copies of the service descriptor with different descriptor-ids. The descriptor-cookie is an optional secret password of 128 bits shared between the hidden service and its clients.

Additionally, the service descriptor contains inter alia the public identity key **permanent-key** of the hidden service, a **secret-id-part** necessary to verify that the service descriptor belongs to the descriptor-id, a timestamp **publication-time** when the descriptor has been created, a list of **introduction-points** and the **signature** of the hidden service descriptor signed with its private identity key. The list of introduction points can contain a single **service-authentication** item, which can be used for the authentication of clients by the hidden service. This is followed by an arbitrary number of introduction points. Each introduction point is identified by the relay's fingerprint and has an IP address plus port, the relay's onion key, the service key generated for the particular introduction point and an **intro-authentication** item. In addition to **service-authentication** client authentication can be performed with the data contained **intro-authentication** as well, but the latter mechanisms is depended on the selected introduction point. The list of introduction points may be encrypted with AES in counter mode and the descriptor-cookie as the secret key. Thus, only clients who know the cookie can decrypt the list in order to contact the hidden service.

Currently, Tor generates two replicas of the service descriptor and publishes each replica to three hidden service directories. Thus, at any time six hidden service directories serve service descriptors for a specific hidden service. The hidden service learns all hidden service directories from the consensus document and sorts their fingerprints in a circular ring. Both replicas are uploaded to the three hidden service directories which fingerprints follow in the ring immediately after the two replicas' descriptor-ids. Service descriptors are uploaded via HTTP once an hour or whenever their content has changed using a circuit to a hidden service directory and the `begin_dir` relay cell mechanism.

Obtain hidden service descriptor

To contact a hidden service a user only needs the `z.onion` address of the hidden service. The value `z` is calculated as the first 80 bits of the SHA-1 hash value of the hidden service identity key encoded as Base32. The Base32 encoding represents the 80 bits as 16 alphanumerical characters [39]. For example, a `.onion` address could look like `3g2upl4pq6kufc4m.onion`. The client can calculate the descriptor-id of the hidden service descriptor based on the `.onion` address, because the descriptor-id is computed with the decoded hash value. In addition, the current timestamp and the replica number are also known. If a descriptor-cookie is used, the user needs to possess this value as well. Then the client can locate the six responsible hidden service directories for the current service descriptor and downloads the descriptor from one of those directories via a Tor circuit after connecting to it with a `begin_dir` relay cell and requesting the service descriptor by its descriptor-id via HTTP.

Establish a rendezvous point

The client chooses randomly a Tor relay as the rendezvous point with the hidden service. It builds a circuit to the selected relay and sends an `establish_rendezvous` relay cell to it, which

consists only of a 20 byte random *rendezvous cookie*. The rendezvous point associates the cookie with the the client circuit and acknowledges the creation of the rendezvous point with an empty `rendezvous_established` relay cell. The rendezvous cookie is later needed to connect the client and the hidden service together at the rendezvous point.

Introduce client to hidden service

The client selects one of the introduction points contained in the service descriptor, builds a circuit to it and sends it an `introduce1` relay cell. This cell contains unencrypted the hash value of the service key of the hidden service. Furthermore, the cell includes a version field set to 3 for the current introduction protocol, optional authentication data of the form TLV, a deprecated timestamp not longer in use, the rendezvous point's IP address, port and onion key, the rendezvous cookie and the first half of a Diffie-Hellman handshake. These fields are all encrypted by the public service key of the hidden service used for the chosen introduction point. The hidden service protocol uses the TAP handshake in order to authenticate the hidden service by the client and to derive session keys shared between the client and the hidden service.

When the introduction point receives an `introduce1` relay cell, it checks whether it recognises the service key contained in it and sends an `introduce2` relay cell along the corresponding circuit to the hidden service. This cell has the encrypted portion of the `introduce1` relay cell as payload. The introduction point acknowledges this to the client with an empty `introduce_ack` relay cell. The hidden service checks that the received `introduce2` relay cell is not a replay by keeping a cache of `introduce2` relay cells it previously received for a specific service key. After that the encrypted payload is decrypted by the hidden service with the private service key in order to learn the selected rendezvous point.

Rendezvous

The hidden service finishes the TAP handshake, builds a circuit to the rendezvous point and sends the rendezvous cookie, the second part of the Diffie-Hellman handshake and the value KH in a `rendezvous1` relay cell to the rendezvous point. The rendezvous point checks whether it has a circuit associated with the rendezvous cookie. In this case it forwards the Diffie-Hellman and KH values in a `rendezvous2` relay cell to the client. The client finishes the handshake and authenticates the hidden service by verifying KH. Both client and hidden service derive the same shared keys as defined by the TAP handshake. They now have established an end-to-end encrypted connection via the rendezvous point, which can be used for further communication.

When the client wants to send a relay cell to the hidden service, it encrypts and authenticates the cell with Kf and Df, respectively, as explained in Subsection 2.3.3 and sends it along the circuit to the rendezvous point. The rendezvous point forwards the cell to the hidden service, which decrypts the cell with Kf and checks its integrity with Df. The hidden service can transmit data to the client in the same way while using Kb and Db instead of Kf and Df. The cells transported on the circuits are encrypted in the same manner as with the ordinary Tor protocol. Cells from the client to the rendezvous point are encrypted three times in addition to the end-to-end encryption with one layer of encryption being removed at every intermediary node. From the rendezvous

point to the hidden service every node adds one layer of encryption. For the opposite direction the processes is reversed.

In order to communicate with the application running at the hidden service the client sends a `begin` relay cell to the hidden service. The hidden service makes a connection to the application as defined in the configuration and replies to the client with a `connected` relay cell. Both the `begin` and the `connected` relay cells do not include any IP addresses, because the client does not know the real IP address of the hidden service and the address must not be exposed to the client with a `connected` relay cell in order to keep the service anonymous. For all further communication between the client and the application via `data` relay cells the hidden service acts as an exit node, except that only connections to the application are allowed. Data received from the hidden service is handed to the application on the client which requested the hidden service.

3.2.2 Access Control with Hidden Services

Access control can be performed at two points in the hidden service protocol. First, when downloading service descriptors with encrypted introduction points. If a client cannot decrypt the introduction points, it cannot connect to the service, because the client does not know the introduction points and it cannot create valid `introduce1` relay cells without knowing the service key. Thus, only clients in possession of the secret decryption key are allowed to access the service. Second, the hidden service can refuse to connect to the client's rendezvous point after it examined the authentication data in `introduce2` relay cells.

Currently, two access control protocols are specified for hidden services: *basic authorisation* and *stealth authorisation* [37]. The difference between those two protocols is the fact that the basic authorisation is suitable for a larger number of users, but does not conceal the activity of the hidden service. Unauthorised users cannot access the service, but can determine that it is operating. With the stealth authorisation unauthorised users cannot discover that a hidden service is active at all. But this protocol is only feasible for a maximum of 16 users. Both protocols can be configured at the Tor instance running as a hidden service transparently to the application. For more details about the configuration see [37].

Basic authorisation

The basic authorisation protocol uses encrypted introduction points for access control. The hidden service creates a single session key to encrypt the introduction points as described in the previous subsection. In addition, the hidden service generates a descriptor cookie for every user, who should be authorised to access the service. Those descriptor cookies are distributed to users outside of Tor and users configure their Tor clients to use this cookie when accessing the hidden service. The hidden service generates a *client id* for every user based on their descriptor cookie and encrypts the session key with each descriptor cookie using AES. The pairs of client id plus encrypted session key are included in the service descriptor in addition to the encrypted introduction points. The list is sorted by the client id and always a multiple of 16 pairs are generated. Fake entries are added if necessary.

A client locates and downloads the service descriptor normally. It finds its entry in the list of authorised clients by searching for its client id, decrypts the session key with its own descriptor cookie and uses the obtained session key to decrypt the introduction points. Then it connects to the hidden service, while including the descriptor cookie in the `introduce1` and `introduce2` relay cells as authentication data. The hidden service verifies the received descriptor cookie. If the client is still authorised to access the hidden service, the service connects to the rendezvous point to finish the connection. Afterwards, client and hidden service communicate without further restrictions.

When a hidden service wants to remove the access of a user, it stops to include the client entry in the service descriptor. Furthermore, the service refuses a connection after receiving the now invalid descriptor cookie with an `introduce2` relay cell. This is necessary, because the client still knows the introduction points of the hidden service, if the service has not changed them after it revoked the user's access.

Stealth authorisation

The stealth authorisation goes one step further than the basic authorisation by hiding the activity of the hidden service to unauthorised users. The idea is to publish a separate service descriptor for each authorised user. For this purpose the hidden service generates an asymmetric *client key* and a symmetric descriptor cookie for every user. The client key is used as a replacement for the hidden service identity key. The client keys and the descriptor cookies are distributed to users outside of Tor. For each user the hidden service creates a service descriptor with the client key and the descriptor cookie as part of the descriptor-id. Furthermore, the introduction points are encrypted with the descriptor cookie as well. Including the descriptor cookie in the descriptor-id ensures that all service descriptors are distributed to different hidden service directories and that only the authorised users can locate and download them. Because of that, unauthorised users cannot identify the activity of a hidden service.

Clients use the client key and the descriptor cookie in order to receive its service descriptor, decrypts the introduction points and include the descriptor cookie in `introduce1` relay cells. The hidden service decides in the same way as with the basic authorisation protocol, if it allows the access of the client after receiving an `introduce2` relay cell. In order to revoke the access for a particular user the hidden service just needs to stop publishing service descriptors for the user and to close down the circuits to the introduction points used for this user. After that, even a formerly authorised user cannot determine that the hidden service is still active.

3.3 Shortcomings and Drawbacks

The basic design of hidden services did not change significantly over the last years since it was initially proposed. Because of that, the design as it is today has several shortcomings and drawbacks, which are described briefly in this section. This includes a description of attacks against hidden services. Some of those limitations and attacks motivated the redesign of Tor hidden services as introduced in the section afterwards. [40] gives a short summary of the limitations of the current hidden service design and points out possible directions for improvements.

The protocol itself and the format of relay cells is not very flexible and extensible and lacks behind the general Tor protocol. For example, the hidden service protocol only supports IPv4 addresses and not IPv6 addresses, and only the TAP handshake and not the newer ntor handshake to authenticate the hidden service. In contrast, the Tor protocol can support future handshake protocols without the need to change the cell format. The hidden service protocol is not easily extensible like this.
Furthermore, hidden services utilises the same cryptographic primitives as the Tor protocol, especially RSA with only 1024 bit keys. Thus, when the Tor protocol is upgraded to stronger, more modern algorithms, hidden services should be upgraded as well. This can be carried out in two independent steps, but needs to be done eventually. Another problem are the hidden service identity keys. Currently, those keys must be kept online at the hidden service, because they are necessary to sign service descriptors. If an attacker can compromise a hidden service and gets hold of the identity key, he can easily impersonate the hidden service. In order to counter this threat identity keys could be stored offline in a similar way as the directory authorities handle their authority identity keys today.

The hidden services' `.onion` addresses have an important property. They are *self-authenticating*. After a client downloaded a service descriptor for a particular hidden service, it can verify that the public identity key contained in the descriptor matches the `.onion` address, because this address encodes the public identity key. For this reason a user can be sure to always connect to the hidden service he actually requested. But this has the great disadvantage that `.onion` addresses are not human memorable and users cannot handle them in the same way as ordinary domain names. This is a severe limitation from an usability perspective, which could pose an obstacle for the adoption of hidden services.

Introduction points are the primary targets of Denial-of-Service attacks against a hidden service, because an attacker cannot attack a hidden service directly. This can make a hidden service temporary unavailable until new introduction points are selected and new service descriptors distributed, if an attacker is able to shut down the currently used introduction points. A possible countermeasure against this type of attack are *valet nodes* [14]. Clients do not contact introduction points directly, but instead use the valet nodes as contact points with the hidden service. With this design clients do not learn the IP addresses of the introduction points. Thus, valet nodes act as a protection layer in front of the introduction points. The idea is to only have a few introduction points but much more valet nodes. An attacker is unable to attack the introduction points directly and unable to shut down all valet nodes.

Hidden services are slow, which has mainly two reasons. First, the establishment of a connection between the client and the hidden service is complex and expensive, because this involves the indirection via an introduction point and a rendezvous point. Second, the final connection between the client and the hidden service consists of six hops. Because many circuits must be built during this process the introduction of the faster ntor handshake may already improve the situation. Additionally, [13] suggests an improvement of the connection establishment based on

the concept of valet nodes. Rendezvous points are removed completely and replaced by valet nodes, which simplifies the set up of a hidden service connection significantly.
In addition to the performance problem hidden services do not scale well. Introduction points are a major bottleneck, because only a few introduction points need to handle all the connection requests to a hidden service. Furthermore, the current hidden service design does not allow load balancing by distributing traffic to different servers with different IP addresses. A connection request is always forwarded to the same hidden service instance.

The first attack able to identify the real IP address of a hidden service is described in [7]. The attack uses only one single Tor node controlled by the attacker to deanonymise the hidden service within minutes to hours. The idea is that the hostile node repeatedly connects to the hidden service and hopes to be picked as the entry node by the hidden service for the circuit from the hidden service to the rendezvous point. The hostile node correlates the traffic it forwards as the Tor node with the traffic generated as the client communicating with the hidden service. If a match can be found, the node is part of the hidden service circuit to the rendezvous point. But the hostile node does not know, if it is the entry node of the circuit or the middle or exit node. In order to determine the IP address of the hidden service the hostile node makes a list of IP addresses connected to the node in the cases of a traffic match. The IP address contained most in the list is most likely the IP address of the hidden service, because when the hostile node acts as the entry node, it is always connected to the same IP address of the hidden service. If its is not the entry node, it is connected to random Tor nodes with different IP addresses. Thus, the IP address of the hidden service is encountered more frequently.
This attack can be accelerated with a second attacker controlled node, which is always chosen as the rendezvous point while communicating with the hidden service. Then the attacker can easily determine, if the first hostile node A is the entry node of the hidden service. If A is directly connected to the IP address of the rendezvous point, A is the exit node. If both hostile nodes are connected directly to the same node B with the same IP address, B is the exit node and A is the middle node. Otherwise, A must be the entry node of the hidden service.
As a countermeasure against this attack guard nodes were introduced. The attack exploits the fact that without guard nodes it is very likely that the hostile node is eventually selected as the entry node of the hidden service for the circuit to the rendezvous point, because every time a new connection is made to the hidden service a new entry node is chosen. Guard nodes effectively defend against this attack, because the entry node is not changed for every new connection.

Hidden service directories are in an interesting position. They can track the popularity of a hidden service by counting the number of requests for the service descriptor of a particular hidden service. Furthermore, if all six hidden service directories for a specific hidden service collaboratively deny to serve service descriptors of that service, they make the service temporarily unavailable to clients. [10] explores these kind of attacks based on hidden service directories.
The main issue is that descriptor-ids are predictable. Knowing the `.onion` address of a hidden service allows everyone to calculate the future descriptor-ids, because the id only depends on the public identity key, a timestamp and the replica number. Because of that, for any future

point in time it can be calculated, which hidden service directory will be responsible for a specific descriptor-id. An attacker can easily generate an identity key for a hidden service directory responsible for a given descriptor-id, because it is not necessary to calculate the identity key exactly, it just needs to be between the descriptor-id and the identity key of the first hidden service directory following the descriptor-id. By exploiting this weakness an attacker can control the six hidden service directories responsible for a descriptor-id of a specific hidden service and for this reason can effectively measure the popularity of the service or deny the access to it.
In addition, [10] shows that it is possible to harvest hidden service descriptors for almost all hidden services within 24 hours with only minimal resources necessary for the attack. The main idea is to run hidden service directories with identity keys, which fall into the gap between consecutive identity keys of two other hidden service directories. By injecting hidden service directories into every second gap of the circular ring of all hidden service directories service descriptors can be gathered. This enumerates all the hidden services currently operating.

3.4 Next-Generation Hidden Services

The Tor developers work on an improved hidden service design in order to modernise hidden services and to fix shortcomings of the old design. The fundamental concept of hidden service directories, introduction points and rendezvous points is not changed, but the details, for instance relay cell formats, are altered. The new design is incompatible to the old one, but allows to use older Tor nodes, which do not understand the new protocol, as introduction and rendezvous points. The current proposal of next-generation hidden services is described in [41]. The design is not finished yet and subject to minor and major changes. Because of that, the following explanations reflect only the current proposal as of the time of writing of this paper.

A major change is the way how keys are handled. The proposal builds on top of the concept of *key blinding*. It exists a public-private master keypair. Using a random value a *blinded keypair* is derived from the master keypair. Everyone who knows the public master key and the random value can derive the blinded public key, but it is impossible to derive the blinded keypair without knowing the master keypair or to derive the master keypair from the blinded keypair. In addition, a signature created with the blinded private key can be verified with the blinded public key as well. This concept is used to keep the hidden service identity key offline. The following public-private keypairs are employed during the new hidden service protocol:

- A long-term *master identity key* to identify the hidden service. The public component is encoded in a `.onion` address. In contrast to the old design the whole public key is encoded making `.onion` addresses much longer. This key can be stored offline and is only used to generate blinded signing keys.
- A *blinded signing key* to sign descriptor signing keys. This key is changed periodically. Everyone who knows the public master identity key and an optional secret, both together called *credential*, can derive the public blinded signing key.
- A *descriptor signing key* to sign service descriptors. In contrast to the master identity key and

blinded signing key the private component of this key must be stored online at the hidden service for signing the descriptors. Descriptor signing keys can be generated in advance similar to the way how directory authorities handle their keys. Thus, an attacker compromising the hidden service can impersonate the service only for a limited time period.

- A short-term *introduction point authentication key* generated for each introduction point used to identify the hidden service to the introduction point. This key has the same functionality as the service key in the old design.
- A short-term *introduction point encryption key* generated for each introduction point used to establish a connection between the client and the hidden service. This key is used analogously to the onion key in the Tor protocol.

In addition, a symmetric *descriptor encryption key* is utilised to encrypt the introduction points in the service descriptor. For symmetric encryption AES in counter mode with 128 bit keys is proposed, for signatures Ed25519 [42] based on elliptic curves.

The next subsection describes how service descriptors are generated, published, stored and received. This is followed by an explanation of the introduction protocol, i.e. the protocol used by clients to contact a hidden service. The end of this section presents the rendezvous protocol between clients and hidden services. These explanations point out how the new hidden service design tries to fix the limitations of the old design and where the two designs differ.

3.4.1 Service Descriptors

Service descriptors are stored in an unpredictable way in order to defend against the attacks described in Section 3.3 caused by the method for storing descriptors in the old design. The new method depends on a time period, a blinded signing key and a shared random value. The length of the time period is determined by a consensus parameter and the default value is 25 hours. For every period a new blinded signing key is generated. In addition, the directory authorities agree upon a shared random value, which is published in the network status document and valid for one time period. This ensures that attackers cannot calculate the location of service descriptors in advance, because they cannot predict the shared random value, and that the hidden service directories for a single hidden service change every time period. Furthermore, a *subcredential* is computed as the hash value of the hidden service's credential and the current blinded signing key. The private blinded signing key is used for signing descriptor signing keys and the public blinded signing key for fetching descriptors from hidden service directories.

The locations for service descriptors change every time period. In order to avoid that all hidden services always upload their new descriptors at the same time at the beginning of each time period the periods overlap. A consensus parameter determines how much two time periods overlap. Hidden services upload their new descriptors during the overlap interval. The exact point in time is calculated based on the new blinded signing key. Because of that, the simultaneous upload of service descriptors is avoided.

The number of service descriptor replicas and the number of hidden service directories storing each replica is defined by consensus parameters. The default values are two replicas and three hidden service directories for each replica, the same as with the old design. For each replica an index is computed based on the current public blinded signing key, the replica number and the current time period. An addition, an index for each hidden service directory is calculated as well depended on the relay's fingerprint, the shared random value and the current time period. Each replica is uploaded anonymously to the three hidden service directories with the index following immediately after the calculated replica index.
Clients can compute the locations of service descriptors as well, because they can derive the currently used blinded signing key from the hidden service's master identity key contained in the `.onion` address and know the current time period as well as the shared random value from the consensus. They download the service descriptor from one of the responsible hidden service directories by making an anonymous circuit to the directory and requesting the descriptor by the current blinded signing key.
If an optional secret is used as part of the hidden service's credential, a client also needs to know this secret in order to derive the blinded signing key. This secret is not part of the `.onion` address. Without knowing the secret a client cannot download the service descriptor of a hidden service. This method can be utilised to gain protection similar to the stealth authorisation in the old hidden service design.

Service descriptor format

The format of a service descriptor consists of two parts: an unencrypted part and an encrypted part. The unencrypted part contains a certificate of the descriptor signing key, which is used to sign the service descriptor. The certificate in turn is signed by the current blinded signing key in order to guarantee that the descriptor was indeed generated by the requested hidden service. Furthermore, the service descriptor contains the current time period and a revision counter. The revision counter allows to update a service descriptor during the same time period. A hidden service directory always replies with the most current service descriptor, i.e. the descriptor with the highest revision counter. The whole service descriptor including the encrypted part is signed with the private descriptor signing key.

The encrypted portion of the service descriptor is encrypted and authenticated with symmetric keys. The keys are derived from the current public blinded signing key and the current subcredential. Thus, only clients in the possession of the hidden service's credential including the secret value can decrypt the service descriptor and contact the hidden service. The encrypted part is appended by a *message authentication code* (MAC), which is used to verify that the encrypted data is not altered. This is basically a hash value of the data calculated using a symmetric authentication key. The data itself consists of a field with supported authentication types and the introduction points of the hidden service. The authentication types specify the methods the hidden service supports to authenticate the client during the introduction protocol. For each introduction point the service descriptor contains the IP address (IPv4 and/or IPv6) plus port as well as the introduction point authentication key and the introduction point encryption key.

In contrast to the service descriptors in the old design the new service descriptors do not contain the long-term identity key of the hidden service. They only contain the blinded signing key. Because of that, with the new design the hidden service directories does not learn `.onion` addresses of hidden services they store the service descriptors for. Thus, it is impossible to track a hidden service and to enumerate `.onion` addresses by harvesting service descriptors.

3.4.2 Introduction Protocol

The general introduction protocol is the same as with the old hidden service design. The hidden service connects to multiple introduction points and advertises them in the service descriptor. A client connects to one of the introduction points and sends to it a connection request. The request is forwarded to the hidden service.

Establish introduction points

In order to establish an introduction point the hidden service creates a circuit to it and sends an `establish_intro` relay cell. This cell contains the introduction point authentication key for this specific introduction point. This has the form TLV. If the type is zero or one, the `establish_intro` relay cell has the format of the old hidden service protocol as previously described. Thus, a relay can distinguish between the old and the new protocol. The values greater than one indicate the new protocol with the value two specifying an Ed25519 authentication key. In addition, this cell can contain a variable number of extensions fields, again of the form TLV. This mechanism can be used to send additional data in `establish_intro` relay cells without the need to modify the general structure. If a relay does not recognise one extension, it just ignores it. Furthermore, the cell contains a MAC which is calculated over the authentication key and all extension fields in order to prevent replay attacks. As the key for the MAC the number KH from the circuit handshake is used. All fields are signed by the introduction point authentication key. An introduction point receiving an `establish_intro` relay cell verifies the fields of the cell as in the old protocol, associates the authentication key with the circuit and acknowledges the establishment of the introduction point with an empty `intro_established` relay cell.

The establishment of an introduction point is exactly the same both in the new and the old protocol. But the new protocol defines a mechanism to manage introduction points as well. If the type of the authentication key in the `establish_intro` relay cell has the value 255, the remainder of the cell contains data for managing the introduction point. This consists of a command type, the length of the command data and the actual data. Currently, only one command is specified, that is *update encryption keys*. This command is utilised to tell the introduction point the introduction point encryption key, which is currently used by the hidden service for this particular introduction point. It must be sent at least once after the establishment of the introduction point in order to set the initial introduction point encryption key. The payload of this command contains a variable number of keys, a monotonically increasing counter and a signature of the data computed with the introduction point authentication key. The purpose of this command is that a hidden service can change its introduction point encryption key without the need for clients to download a new service descriptor. Instead the hidden service tells the introduction point the new key, which in turn tells this key to a client if necessary.

Introduce client to hidden service

A client needs to know at least one introduction point plus the corresponding introduction point authentication key and introduction point encryption key in order to build a connection to a hidden service. The `introduce1` relay cell sent from the client to the introduction point contains two identifiers for both keys, extension fields in the same way as `establish_intro` relay cells and an additional encrypted payload. The introduction point determines whether the key identifiers in the cell match the keys supplied by the hidden service. In the positive case the cell is forwarded to the hidden service as an `introduce2` relay cell with exactly the same payload as the `introduce1` relay cell. The introduction point acknowledges the successful introduction request with an `introduce_ack` relay cell to the client. This relay cell has only one field with a status value and optional extension fields. If the introduction point does not recognise the introduction point encryption key in the `introduce1` relay cell, it sends the current key belonging to the hidden service to the client as an extension field of the `introduce_ack` relay cell. The introduction point transfers the key as received from the hidden service via the *update encryption keys* command.

In order to establish an end-to-end encrypted connection between the client and the hidden service the new hidden service protocol employs a variant of the ntor handshake using the introduction point encryption key during the handshake. In addition to the normal ntor handshake two additional single-use keys are generated during this process. Those two keys are used to encrypt and authenticate the additional payload of `introduce1` and `introduce2` relay cells. See [41] for all the details of the extended ntor handshake. The additional payload contains the public key used as the first half of the ntor handshake, the encrypted data for the hidden service and a MAC of the whole relay cell payload including the the encrypted data. The MAC ensures that the payload is received unmodified by the hidden service. This authentication mechanism is an improvement in the new design, because it is not performed in the old hidden service protocol. The data is encrypted and authenticated with the two keys generated during the extended ntor handshake.

The encrypted data for the hidden service consists of the following information: the rendezvous cookie, optional extension fields, the rendezvous point's onion key (either TAP or ntor) and link specifiers to contact the rendezvous point. The link specifiers have the same format as in `extend2` relay cells, including IPv4 and IPv6 addresses. When a hidden service receives an `introduce2` relay cell, it checks if the cell contains the correct introduction point authentication key and introduction point encryption key, reads the public key of the client, performs its part of the extended ntor handshake, authenticates the relay cell and encrypts the additional payload. Then the hidden service has all information necessary to continue with the rendezvous protocol.

Access control during introduction

The new hidden service protocol specifies an access control mechanism, which enforces access control during the introduction protocol, as a replacement for the basic and stealth authorisation protocols of the old hidden service design. This provides more fine-grained access control

as just using the credential mechanism, because access control can be defined on a per-user basis and does not require the distribution of service descriptors for access control. Currently, two mechanisms are specified: *password-based authentication* and *Ed25519-based authentication*. Both mechanisms transport data as an extension field in the encrypted part of `introduce1` and `introduce2` relay cells. The password-based authentication sends a username and a password to the hidden service, which checks these values against a list of authorised users.
With the Ed25519-based authentication every authorised user has a public-private Ed25519 keypair. The client uses the private key to sign a message to the hidden service. This message contains a random value, the public key and the signature. The signature is computed over the random value, the public key and the two identifiers of the introduction point authentication key and the introduction point encryption key contained in the `introduce1/2` relay cells. The hidden service checks whether it recognises the public key as belonging to an authorised user and whether the signature is correct. This authenticates the user. With both mechanisms the hidden service does not connect to the rendezvous point, if the authentication fails. If the authentication succeeds, it continues normally with the rendezvous protocol.

3.4.3 Rendezvous Protocol

The new rendezvous protocol is not very different form the old protocol. In order to establish a rendezvous point a client sends an `establish_rendezvous` relay cell to a relay, which only contains the rendezvous cookie. The relay answers with an empty `rendezvous_established` relay cell. The hidden service connects to a rendezvous point with a `rendezvous1` relay cell consisting of the rendezvous cookie and the server's half of the ntor handshake. The rendezvous point forwards the cell as a `rendezvous2` relay cell to the client with the content unchanged, if it has a circuit associated with the rendezvous cookie provided by the hidden service. Because only the handshake data included in `rendezvous1/2` relay cells has changed in the new protocol, relays, who do not understand the new hidden service protocol, can still be selected as rendezvous points, because they do not need to process the handshake data. When the connection between the client and the hidden service is established, they communicate in the same way as in the old hidden service protocol. The only difference is that they use the extended ntor handshake to derive the session keys.

3.5 Open Research Questions

The new hidden service design mainly upgrades the hidden service protocol to the same standard as the main Tor protocol. It now supports IPv6 addresses, the new ntor handshake and improves the protocol reflecting up-to-date cryptography best practices. Especially, the handling and the security of hidden service identity keys is improved by making it possible to store them offline. In addition, the relay cell formats are made extensible in order to allow future enhancements of the protocol without necessarily changing the format and a large redeployment of the Tor software. Furthermore, the new unpredictable design for storing service descriptors effectively defends against the hidden service directory attacks presented in [10].
Additionally, the new design enforces access control mainly during the introduction protocol, whereas the authorisation protocols in the old hidden service design are based on the encryption

of introduction points in service descriptors. The new approach has the advantage that adding, changing or removing access rights can be performed locally at the hidden service without the need to redistribute service descriptors. The protocol also allows to easily add further access control mechanisms in addition to the password-based and Ed25519-based methods. However, there is no equivalent to the stealth authorisation protocol specified yet. But such a mechanism can be implemented with hidden service credentials and a secret value part of the credential.

But the new hidden service design is not fully finished yet. Some parts need more research and thinking, whereas other parts need a more thorough specification. One point is that the relay cells used during the rendezvous protocol are not extensible in the same manner the relay cells used for the introduction protocol. Making them extensible would allow future amendments to this part of the protocol as well. In addition, the idea of updating the introduction point encryption keys needs more analysis. Is this mechanism worth the complexity? What are the benefits? What are the drawbacks? Or should clients just download a new service descriptor, if they only possess an expired key?

Furthermore, some major components of the redesign are not yet fully discussed and specified, for example the scaling of hidden services. The new design should support hidden services, which use more than one node for the service in order to better handle high usage load on the hidden service. The Tor developers investigate two major alternative designs for scaling hidden services [41]. With the first option each hidden service node builds a circuit to each introduction point and the introduction points distribute the requests to the different hidden service nodes. One master node is responsible for selecting the introduction points and for publishing the service descriptor. In the alternative design every hidden service node picks its own introduction points independently and a master node combines all introduction points into one single service descriptor. In this scenario the clients choose the introduction point and with it balance the load over all hidden service nodes. Both options need a mechanism to select another master node in order to avoid an inoperable service in the case a master node shuts down.

Another questions is how a hidden service chooses its introduction points. This can be boiled down to three major questions [38]. First, how many introduction points should a hidden service select? Is the current formula used for estimating the popularity of a hidden service working? What is the trade-off between the number of introduction points, load on the network for maintaining them and security? Second, which relays can be introduction points? Currently, every relay with the *stable* flag can be selected as an introduction point. Third, what is the lifetime of an introduction point? After what time period should new introduction points be selected? In the present design introduction points are rotated between 18 to 24 hours. But the recent research about entry guards suggests that a longer period is beneficial for security, because it reduces the likelihood that an attacker controlled relay is picked as an introduction point.

A few limitations of the old hidden service design are not tackled by the new design at all or not specifically. The resistance of Denial-of-Service attacks against introduction points is not

improved. Introduction points still face the same risk of DoS attacks. If the lifetime of an introduction point is increased, this may be even a greater problem, because an attacker can make the hidden service unavailable for a longer period of time by shutting down the introduction points. If unavailable introduction points are rotated, an attacker can use this attack to force the hidden service to change its introduction points very frequently, which increases the likelihood that the hidden service picks an attacker controlled introduction point. But it remains to been shown that this is a realistic attack.
Furthermore, the performance problems of hidden services are not specifically addressed. By moving to the ntor handshake and elliptic curve public-key cryptography the situation could already improve, but the consequences will only be evident once the new design is implemented and deployed. At the moment the general issue of the complex connection set up with the hidden service and the final six-hop connection is not attacked by the new design. In addition, the problem of non-memorable `.onion` addresses is not solved as well. In contrast, with the longer `.onion` addresses in the new design the situation is even worse. Because of that, solving this problem is more urgent than ever.

Tor location-hidden services provide anonymity to the service operator in addition to the anonymity given to the user by Tor. The new design of hidden services is necessary in order to modernise hidden services, to make them more secure and future-proof. The redesign does not solve all limitations of hidden services, but it will certainly make improvements easier in the future.

4 Conclusion

Anonymity is an important property in information security. In contrast to confidentiality the goal is to hide the communication partners, who is communicating with whom, not only the communication content. But this cannot be achieved by the communication partners alone, they need cover traffic to hide their own traffic in. Because of that, an anonymity system can only provide sufficient protections to its users, if it has many diverse groups of users using and contributing to the system. This places anonymity systems in an area of conflict, where they have to balance anonymity, security and usability.

This paper introduces Tor as one example of an anonymity system, which is utilised to provide anonymous communications for interactive applications such as Web browsing or instant messaging. Tor cannot only provide anonymity to users, who access non-anonymous services like public Web sites, but also to service operators in the form of location-hidden services, which makes it possible to have a complete anonymous communication between a user and a service. The general mode of operation of Tor and hidden services in described as well as the details of the protocols explained in-depth. This includes limitations, shortcomings and drawbacks in order to provide the reader with a thorough understand of Tor, empowers the reader to critically discuss Tor and to comprehend the situations in which Tor cannot provide anonymity.
A special focus is laid on location-hidden services, which are currently subject to a major overhaul. The current hidden service design is described and its shortcomings and drawbacks analysed. Keeping this in mind the current proposal of the next-generation hidden services is presented as well. This concludes with a list of open research questions, which are not yet answered by the current proposal.

Tor is an important piece of software, which helps million of people every day to stay safe online and to communicate anonymously. It is also a very interesting project to conduct research, because all the development is carried out in public and the source code and specifications are publicly available as well. This paper helps everyone, who wants to learn more about Tor, to start this journey by explaining the inner workings of Tor in detail. Hopefully this will enable people willingly to contribute to Tor to actually doing it.

Bibliography

[1] Dingledine, R., Mathewson, N., & Syverson, P. August 2004. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*.

[2] Dingledine, R., Mathewson, N., & Syverson, P. Challenges in deploying low-latency anonymity. Technical Report 2005-02-001, The Tor Project, February 2005.

[3] Dingledine, R. & Mathewson, N. Design of a blocking-resistant anonymity system. Technical Report 2006-11-001, The Tor Project, November 2006.

[4] Perry, M., Clark, E., & Murdoch, S. March 2013. The Design and Implementation of the Tor Browser [DRAFT]. `https://www.torproject.org/projects/torbrowser/design/`. Last Accessed 2014-09-05.

[5] Murdoch, S. J. & Danezis, G. May 2005. Low-Cost Traffic Analysis of Tor. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*. IEEE CS.

[6] Johnson, A., Wacek, C., Jansen, R., Sherr, M., & Syverson, P. November 2013. Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries. In *Proceedings of the 20th ACM conference on Computer and Communications Security (CCS 2013)*. ACM Press.

[7] Øverlier, L. & Syverson, P. May 2006. Locating Hidden Servers. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE CS.

[8] Murdoch, S. J. November 2006. Hot or Not: Revealing Hidden Services by their Clock Skew. In *13th ACM Conference on Computer and Communications Security (CCS 2006)*, 27–36. ACM Press.

[9] Zander, S. & Murdoch, S. J. July 2008. An Improved Clock-skew Measurement Technique for Revealing Hidden Services. In *Proceedings of the 17th USENIX Security Symposium*, 211–226.

[10] Biryukov, A., Pustogarov, I., & Weinmann, R.-P. May 2013. Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 80–94.

[11] Loesing, K., Sandmann, W., Wilms, C., & Wirtz, G. July 2008. Performance Measurements and Statistics of Tor Hidden Services. In *Proceedings of the 2008 International Symposium on Applications and the Internet (SAINT)*. IEEE CS.

[12] Lenhard, J., Loesing, K., & Wirtz, G. June 2009. Performance Measurements of Tor Hidden Services in Low-Bandwidth Access Networks. In *Proceedings of the 7th International Conference on Applied Cryptography and Network Security (ACNS 09)*, Abdalla, M., Pointcheval,

D., Fouque, P.-A., & Vergnaud, D., eds, volume 5536 of *Lecture Notes in Computer Science*, 324–341. Springer-Verlag.

[13] Øverlier, L. & Syverson, P. June 2007. Improving efficiency and simplicity of Tor circuit establishment and hidden services. In *Proceedings of the Seventh Workshop on Privacy Enhancing Technologies (PET 2007)*, Borisov, N. & Golle, P., eds. Springer-Verlag.

[14] Øverlier, L. & Syverson, P. June 2006. Valet Services: Improving Hidden Servers with a Personal Touch. In *Proceedings of the Sixth Workshop on Privacy Enhancing Technologies (PET 2006)*, Danezis, G. & Golle, P., eds, 223–244. Springer-Verlag.

[15] Hopper, N. March 2014. Challenges in protecting Tor hidden services from botnet abuse. In *Proceedings of Financial Cryptography and Data Security (FC'14)*.

[16] Acquisti, A., Dingledine, R., & Syverson, P. January 2003. On the Economics of Anonymity. In *Proceedings of Financial Cryptography (FC '03)*, Wright, R. N., ed, volume 2742 of *Lecture Notes in Computer Science*. Springer-Verlag.

[17] Dingledine, R. & Mathewson, N. June 2006. Anonymity Loves Company: Usability and the Network Effect. In *Proceedings of the Fifth Workshop on the Economics of Information Security (WEIS 2006)*, Anderson, R., ed.

[18] Chaum, D. February 1981. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2).

[19] Gollmann, D. 2011. *Computer Security*. John Wiley & Sons, 3rd edition.

[20] Reiter, M. & Rubin, A. June 1998. Crowds: Anonymity for Web Transactions. *ACM Transactions on Information and System Security*, 1(1).

[21] Leech, M., Ganis, M., Lee, Y.-D., Kuris, R., Koblas, D., & Jones, L. March 1996. SOCKS Protocol Version 5. RFC 1928 (Proposed Standard).

[22] The Tor Project. Tor directory protocol, version 3. `https://gitweb.torproject.org/torspec.git/blob/HEAD:/dir-spec.txt`. Last Accessed 2014-09-13.

[23] Dingledine, R. September 2013. The lifecycle of a new relay. `https://blog.torproject.org/blog/lifecycle-of-a-new-relay`. Last Accessed 2014-09-10.

[24] Perry, M. June 2010. Tips for Running an Exit Node with Minimal Harassment. `https://blog.torproject.org/blog/tips-running-exit-node-minimal-harassment`. Last Accessed 2014-09-10.

[25] Loesing, K. Overview of Statistical Data in the Tor Network. Technical Report 2011-03-001, The Tor Project, March 2011.

[26] Loesing, K., Perry, M., & Gibson, A. Bandwidth Scanner specification. `https://gitweb.torproject.org/torflow.git/blob/HEAD:/NetworkScanners/BwAuthority/README.spec.txt`. Last Accessed 2014-09-12.

[27] Dingledine, R. & Mathewson, N. Tor Path Specification. https://gitweb.torproject.org/torspec.git/blob/HEAD:/path-spec.txt. Last Accessed 2014-09-14.

[28] Dingledine, R. October 2013. Improving Tor's anonymity by changing guard parameters. https://blog.torproject.org/blog/improving-tors-anonymity-changing-guard-parameters. Last Accessed 2014-09-14.

[29] Kadianakis, G. & Hopper, N. The move to a single guard node. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/236-single-guard-node.txt. Last Accessed 2014-09-14.

[30] Dingledine, R. & Mathewson, N. Tor Protocol Specification. https://gitweb.torproject.org/torspec.git/blob/HEAD:/tor-spec.txt. Last Accessed 2014-09-19.

[31] Diffie, W. & Hellman, M. E. November 1976. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6), 644–654.

[32] Goldberg, I., Stebila, D., & Ustaoglu, B. January 2013. Anonymity and one-way authentication in key exchange protocols. *Designs, Codes and Cryptography*, 67(2), 245–269.

[33] Bernstein, D. J. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006*, Yung, M., Dodis, Y., Kiayias, A., & Malkin, T., eds, volume 3958 of *Lecture Notes in Computer Science*, 207–228. Springer-Verlag.

[34] Manils, P., Chaabane, A., Le Blond, S., Kaafar, M. A., Castelluccia, C., Legout, A., & Dabbous, W. April 2010. Compromising Tor Anonymity – Exploiting P2P Information Leakage.

[35] Mathewson, N. Two improved relay encryption protocols for Tor cells. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/202-improved-relay-crypto.txt. Last Accessed 2014-09-20.

[36] Mathewson, N. Migrate server identity keys to Ed25519. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/220-ecc-id-keys.txt. Last Accessed 2014-09-20.

[37] The Tor Project. Tor Rendezvous Specification. https://gitweb.torproject.org/torspec.git/blob/HEAD:/rend-spec.txt. Last Accessed 2014-09-26.

[38] Kadianakis, G. August 2014. [tor-dev] On picking Introduction Points in Next Generation Hidden Services. https://lists.torproject.org/pipermail/tor-dev/2014-August/007335.html. Last Accessed 2014-10-03.

[39] Josefsson, S. October 2006. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard).

[40] N.N. April 2013. Hidden Services need some love. https://blog.torproject.org/blog/hidden-services-need-some-love. Last Accessed 2014-09-27.

[41] Mathewson, N. Next-Generation Hidden Services in Tor. `https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/224-rend-spec-ng.txt`. Last Accessed 2014-10-03.

[42] Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., & Yang, B.-Y. September 2012. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2), 77–89.