JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Thomas Raab, BSc

Submitted at
Institute of Networks and Security

Thesis Supervisor
Univ. -Prof. Priv.-Doz. Dr. René Mayrhofer

April 2021

Unlinkable Onion Services: Improved Resilience against Metadata Analysis

Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Sworn Declaration

Kurzfassung

In unserer digitialisierten Gesellschaft, in der verschiedenste Interessengruppen versuchen die Internetnutzung zu kontrollieren und zu überwachen, ist Anonymität eine der wesentlichen Eigenschaften um Privatsphäre im Internet zu gewährleisten. Eine der Techniken, die zur Wahrung dieser Eigenschaft eingesetzt wird, ist das Anonymisierungsnetzwerk *Tor*, welches die Verbindungsdaten einer Kommunikation so verschleiert, dass deren Initiator nicht mehr identifiziert werden kann. Da dies aber nur den Initiator, nicht aber weitere Kommunikationsteilnehmer schützt, wurden Tor *Onion Services* entwickelt, welche die Anonymität des Senders und Empfängers sicherstellen. Bedingt durch die Metadaten die bei der Nutzung dieser Onion Services entstehen, könnten Angreifer durch die Nutzung zusätzlicher Informationsquellen dennoch in der Lage sein, die Teilnehmer einer Kommunikation zu identifizieren.

Im Zuge dieser Arbeit wurde ein Protokoll entwickelt, welches Metadaten, die zur Identifikation von Kommunikationsteilnehmern führen können, weitestgehend reduziert. Dazu wird ein zweistufiges Addressierungsschema eingesetzt, dass Benutzern ermöglicht, eine individuelle Adresse für einen Service über über eine öffentliche Service Adresse zu erhalten, deren Verwendung nicht zurückverfolgt werden kann. Um die technische Machbarkeit dieses Protokolls unter Beweis zu stellen, wurde ein Prototyp auf Basis von Python entwickelt. Da die Latenzzeit eines Services eine der ausschlaggebenden Kriterien in der Nutzungsentscheidung darstellt, wurde eine Performance-Analyse durchgeführt, um die Provisionzeit von Onion Services zu messen, da diese einen maßgeblichen Einfluss auf die Dauer der Adressausstellung hat. Die Architektur und Vorgehensweise dazu musste eigens konzipiert und implementiert werden, da zum Zeitpunkt der Erstellung dieser Arbeit keinerlei Forschung zur Provisionzeit von Onion Services in ihrer aktuellen Version existierte.

Eine statistische Analyse der Ergebnisse zeigte, dass die Dauer der Ausstellung individueller Adressen mithilfe des entwickelten Protokolls die Akzeptanzgrenze von Benutzern mit 6.35 Sekunden überschreitet. Das trifft jedoch nicht auf den Service-Zugriff unter Verwendung der individuellen Adresse zu, was impliziert, dass die Verwendung des Protokolls nach einer Verbesserung des Adressaustellungsmechanismus möglich ist. Das würde die Metadaten beim Zugriff auf einen Onion Service reduzieren und somit zur Verbesserung der Anonymität der Kommunikationsteilnehmer beitragen.

Abstract

In our digitized society, in which different organizations attempt to control and monitor Internet use, anonymity is one of the most desired properties that ensures privacy on the Internet. One of the technologies that can be used to provide anonymity is the anonymization network *Tor*, which obfuscates the connection data of communications in a way that its initiator cannot be identified. However, since this only protects the initiator without protecting further communication participants, Tor *Onion Services* were developed, which ensure the anonymity of both the sender and the recipient. Due to the metadata created when using these Onion Services, adversaries could still be able to identify participants in a communication by using additional sources of information.

In the course of this thesis, a protocol was developed that reduces metadata leading to the identification of communication participants as far as possible. For this purpose, a two-staged addressing scheme was employed that allows users to obtain an individual address for a service via its public service address, which cannot be traced back. To prove its technical feasibility, a prototype of the protocol was implemented based on Python. Since latency is one of the decisive criteria in the usage decision of services, a performance analysis was carried out to measure the provisioning time of onion services, since this has a significant influence on the duration of address issuing. The architecture and procedure for this had to be specially designed and implemented, as at the time of writing no research existed on the provisioning time of onion services in their current version.

A statistical analysis of the results revealed that the duration of issuing individual addresses using the proposed protocol exceeds the acceptance threshold of users with 6.35 seconds. However, this does not apply to service access using the individual address, implying that the use of the protocol is possible after improving the address issuance procedure. This would reduce the metadata when accessing an Onion service and thus help improve the anonymity of communication participants.

1 Introduction

Communication is the cornerstone of our modern society. Although the demand for communication has not changed, the technological innovations in the last three decades caused major changes in the nature of communication. Messages and data are now no longer transmitted in analogue form, but exchanged via the Internet. The convenience and usefulness of the Internet made it increasingly popular among the population, allowing it to become an integral part of both the private and the professional daily life [1].

The foundation of communication over the Internet is the *TCP/IP* stack, which is a collection of network protocols that enable end-to-end communication between network participants. The stack is composed of four layers that describe the physical data transmission (Link Layer), the forwarding and routing of packets (Internet Layer), the end-to-end communication (Transport Layer) and the exchange of application-specific data (Application Layer) [2, pp. 45-48]. The Internet Protocol (IP) is one of the eponymous protocols of this stack, whose function is the routing of packets across network boundaries. Routing is often performed by independent devices, which are referred to as routers. In order for the routers to determine where to forward a packet, each IP packet contains an IP header in addition to the payload data. In order to make a routing decision, the routers must know the destination of the IP packet. Apart from the sender address and other routing-related information, this recipient address is part of the IP header. Due to the stateless concept of IP, in order to enable reliable data transmission, the Transmission Control Protocol (TCP) is used to transfer data between applications. Besides partitioning the byte stream to be sent into segments, TCP offers error detection as well as flow control and ensures that the segments sent are eventually reassembled in the correct order [3].

In case an IP packet is transmitted without any protection, every router on the path from the sender to the recipient would theoretically be able to inspect and manipulate its content. To overcome this issue, an encryption protocol such as Transport Layer Security (TLS) can be applied to TCP segments, which provides authenticity, integrity and confidentiality for data transmission. Since TLS operates on a higher layer in the TCP/IP stack compared to IP and the IP address of the recipient is needed for the routing procedure, the IP header is not protected by TLS. At first glance, this may not seem like an issue, as the actual content of the message is encrypted, nevertheless, the following information can be obtained from the non-encrypted metadata of the communication: [2, pp. 853-857]

- Both the sender and the recipient of the communication can be identified from the IP header. Although it is not possible to determine what is being communicated, it is possible to discover who is communicating with whom.
- The TCP header, which is also not encrypted when using TLS, reveals the source and destination port of the communication. This allows conclusions to be drawn about the application that is being used in the communication (e.g. port 443 indicates that a website is accessed).

- The frequency and time of communication can be determined from the IP header and the intercepting system.

For example, a company could use this information to monitor whether employees are browsing online job portals or social media during working hours, provided that only one website is running on the monitored servers at a time or that Server Name Indication (SNI) is available. Another example are state actors that could use the meta-information provided in the IP and TCP header to enforce censorship and oppress their opponents' freedom of speech by disturbing or preventing access to their means of communication, e.g. Internet forums.

In order to protect not only the payload but also the metadata of the communication and thus the communication participants, anonymization services like the *Tor* network were developed [4]. Tor uses a concept referred to as *Onion Routing* to disguise the identity of the client based on TCP. A schematic illustration of general functioning of Onion Routing is shown in Figure 1.1.

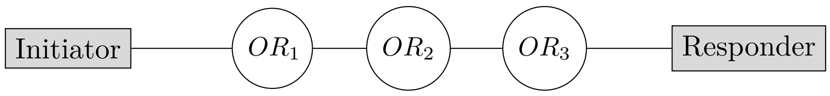


Figure 1.1: Tor Onion Routing Network

Using Onion Routing, a message is not sent directly to the recipient, but routed through several randomly selected Onion Routers (OR). In order to prevent an OR from accessing the data of the TCP segment, the initiator establishes a shared key with each OR in advance. It encrypts the segment with the shared keys to generate a layered encrypted object that conceptually resembles an onion, due to which this procedure is named. On the path from the initiator to the responder, each OR decrypts one layer with its shared key. The decrypted data contains information on the next hop in the path, which implies that each OR only knows its predecessor and successor. The reply of the responder is encrypted by each OR with its shared key with the initiator, so that ultimately only the initiator can decrypt the response. [4]

Onion Routing protects only the anonymity of the initiator, but not that of the responder, whose IP address must be known to enable communication. To overcome this issue, Tor onion service were developed to ensure both initiator and responder anonymity within the Tor network. In order to achieve this level of anonymity, services are no longer addressed through their IP address, but through an *onion address*. This address is based on the key material that is generated for an onion service during its creation. Each service chooses several *Introduction Points* (IPs), which are the first point of contact for communication with the onion service. Information about these IPs and further service-related data are consolidated in service descriptors. These descriptors are uploaded to specific *Hidden Service Directories* (HSDirs) depending on the initial created key material and the time of upload. A user who wants to access the service needs to know its onion address. With this address, the user can download the service descriptors from the HSDirs, which contain the addresses of the IPs. Once the user obtains this information, they choose a random node in the Tor network that will acts as its *Rendezvous Point* (RP) and establishes a connection to it. In principle, this RP eventually becomes the middleman between the service and the user. Subsequently, the user contacts one of the IPs and provides the chosen RP. The IP relays the request to the service, which in turn decides whether it wants to communicate with the client. If yes, it connects to the RP provided by the client, so that both client and service are now connected to the RP, which enables communication between client and service.

1.1 Problem Statement

Since the Tor network is run by volunteers, anyone can contribute computational resources in the form of computers or servers, that can eventually become RP, IP or HSDir of an onion service. This does not seem like an issue at first glance, because even if an adversary were to control one of the onion service components, it would only be able to collect minimal amounts of metadata that on their own can be linked to neither a server nor a client.

However, the situation is different if an adversary is able to obtain additional metadata from another source and correlate this data with the metadata of the onion service. This scenario is particularly effective if an adversary can control one of the HSDirs of a service. Since the descriptors are stored encrypted on the HSDir, no information can be gained from them, but it is possible for the adversary to determine when and how often a particular onion service is accessed, because it can monitor the HTTP requests associated with the descriptor downloads. If the adversary is able to establish a temporal link between the download time of the descriptor from the HSDir and other information such as Tor traffic on the corporate network the user is connected to, it may be possible to identify the user, or at least the device that submitted the HTTP request.

1.2 Objectives

The goal of this thesis is the proposal of a novel protocol that allows the access to an onion service, while preventing its linkability through HSDir descriptor downloads. In order to demonstrate its technical feasibility, a Proof of Concept (PoC) shall be implemented, that enables clients to communicate with a Hypertext Transfer Protocol (HTTP) server. In order to assess whether the proposed protocol is usable in practice, a performance analysis shall be carried out to measure the time it takes to provisioning an onion service, as this will be an integral part of the proposed protocol. Thereby, the different variants of onion service shall be compared. The results of the measurements shall be examined by means of a thorough statistical analysis to uncover any correlations. Using the results from the prototype implementation and the performance analysis, it shall finally be determined whether the developed protocol can reduce the metadata of an onion service access with reasonable additional latency.

2 Related Work

Since the research in the area of onion services is very focused, the related work section is structured according to the two main components of this thesis.

2.1 Performance Measurement

Köpsell et al. [5] discovered that an increase of delay in terms of latency in anonymization services is linearly related to the drop-out rate of users, as less security-affine participants question the utility of this service and stop using it. This implies that the number of users decreases as the delay in the anonymity service increases, which has a negative impact on the quantity of anonymity provided by the service. Besides usability, this is the major motivation for conducting performance measurements in anonymization services. The types of performance measurements can roughly be divided into measurements of the performance of the Tor network and measurements of the performance of onion services.

2.1.1 Tor Performance

Wendolsky et al. [6] compared the performance of the two low-latency anonymization services *Tor* and *AN.ON* regarding the performance indicators latency and bandwidth. Based on the work of Köpsell et al. [5] they conclude that the tolerance limit for latency is four seconds for inexperienced users. This outcome can only partially be used for comparison with the results of this thesis, because the architecture of Onion services is much more complex and involves more nodes. Furthermore, Wendolsky et al. [6] noticed performance variations in the use of Tor depending on the time of day.

To measure long-term performance trends in the Tor network, Custura et al. [7] developed *OnionPerf*. This software tool measures Tor network performance by tracking the bulk download of random data produced by traffic generators. This random data is either hosted on a local machine to emulate access via the Internet, or the data is hosted via onion service to simulate access to an onion service [8]. In contrast to the previously described work, Panchenko et al. [9] focused their study on the examination of node limitations and their relations to latency and throughput. They carried out several experiments in a private tor network as this represents an optimal environment without external disruptive influences. One of their findings indicates that the more clients use a node to build a circuit, the lower is its throughput. An interesting fact is that the throughput remains constant from about 14 clients on, no matter how many circuits the node is involved in.

Furthermore, Panchenko et al. [9] analysed the effect of different parameters of path selection on the *Round Trip Time* (RTT) of a circuit. The shortest mean RTT is achieved when the selection of nodes on the path is not carried out uniformly, but weighted according to the advertised bandwidth of the nodes. Moreover, the study indicates that the RTT of a circuit also depends on the geographical diversity of the nodes in the circuit.

2.1.2 Onion Service Performance

Lösing et al. [10] were the first to analyse the performance of onion services in terms of latency in a scientific publication. In this paper, a special focus is laid on overall response time, as the work of Köpsell et al. [5] suggests that latencies and connection setup times have a much greater effect on service usability than bandwidth. For the measurements, two Tor instances are used. The first instance serves as an onion service, while the other instance is configured as a relay, acting as an *Introduction Point* (IP). Changes in the source code ensure that both the onion service and the client use the IP controlled by the authors. The *Rendezvous Point* (RP) is not controlled by the authors because the selection of a specific RP does not work in case of circuit cannibalization. Clients for accessing the onion service were periodically created to avoid any caching of the *Onion Proxy* (OP) on the client. Every client performs only a single access-attempt before it is discarded. The mean overall response time for the measurements was about 24 seconds. Considering the overall duration in relation to the individual sub-steps for accessing an onion service, it is noticeable that building the circuits from the client to the introduction point and fetching the service descriptor takes almost half of the overall duration.

Similarly to Lösing et al. [10], Wilms [11] measures the overall response time of the access to an onion service by controlling the individual instances that are needed to operate the service. In contradiction to Lösing et al. [10], Wilms [11] found a way of also controlling the RP. Despite the *RendNodes* option in the Tor client configuration allowing the selection of a specific RP, a different RP was previously selected because three internal three-hop circuits are generated during bootstrapping, which are cannibalized when the connection to the RP is established. Since these internal circuits were built without considering the desired RP, a different RP was chosen. By changing the Tor source code, an internal circuit with the RP as last node is built at bootstrapping, which is then cannibalized as RP circuit. The RP, the onion service and the OP are operated on the same machine, as no direct communication between the components can introduce bias to the connection times. Wilms [12] use the tool *PuppeTor* [13] to deploy the Tor clients and measure the overall response time, which was on average about 39 seconds. This is considerably longer than the overall response time measured by Lösing et al. [10]. However, the reason for this increase could not be determined.

Lehnhard et al. [14] focus their work on measuring the performance of onion services in low-bandwidth access networks, because the communication overhead of anonymization services becomes even more apparent in this context. As in the work of Wilms [11], the IP, RP and Onion services are controlled by the authors and hosted on the same server. In the first measurement phase the client uses a low-bandwidth access network (analog modulation via the telephone network and *Enhanced Data Rates for GSM Evolution* (EDGE)), while the server uses a broadband network access. In the second measurement phase, this concept is reversed, such that the measurement server hosting the RP, IP and the onion service uses the low-bandwidth access network and the clients uses the broadband network. Similar to the other studies, new clients for access to onion services are regularly created, which are discarded after one access attempt in each case, in order to avoid incorrect measurements through caching. In addition to measuring the overall response time, a measurement of the bootstrap time of each Tor client instance is carried out. The measurement results show that the bootstrap time is a major problem when accessing onion services from a low-bandwidth access network, since it takes about five times longer (232.9 seconds for EDGE and 249.0 seconds for modem compared to 22.9 seconds for broadband) than using a broadband network. Considering the overall response time, the access of a client to an onion service using a low-bandwidth access network takes about twice as long as using a broadband network. In the reversed scenario where the server is using the low-bandwidth access network, the difference shrinks to eight seconds between EDGE and broadband and only one second between broadband and modem. The analysis of the circuit creation times shows that they represent a bottle-neck when accessing onion services, especially when using a low-bandwidth access network.

There is currently little to no research on the provisioning time of onion services. Furthermore, all the above mentioned work is based on the outdated onion service V2 protocol, which will no longer be supported in the foreseeable future [15]. This work fills this gap by providing a comprehensive performance analysis of the provisioning of a V3 onion service including all necessary sub-steps.

2.2 Unlinkability

Oeverlier et al. [16] were the first to publish an attack on the anonymity of onion services. For this attack it is assumed that the attacker controls both a relay in the network and a client of an onion service. Any relay in the Tor network that offers stability can be used by the onion service to build a circuit to the RP. The attacker creates several rendezvous circuits to the onion service and sends a specific traffic pattern over these circuits. With the help of logged time values and direction information from the client and the relay, the attacker can determine whether his relay is part of a circuit to the onion service. If this is the case, the attacker can further ascertain at which position his node is located in this circuit. Since the attacker has selected the RP, he knows its address. This enables him to determine if his node is closest to the RP. If the node has unknown addresses on each side, it can be either the middle node in the circuit or the first node of the path to the onion service. This distinction can be made using a timing analysis or the *predecessor attack* [17]. If the attacker's node is the first node on the path to the onion service, the attacker has the IP address of the onion service and thus can conclude its position. To prevent this type of attack, *Guard nodes* have been introduced, which are described in section 3.2.5.

Biryukov et al. [18] focus on the discovery of onion services through the collection of onion service descriptors. They claim to have collected all service descriptors available at the time within two days at the cost of only 100 USD. To collect the service descriptors of all onion services, it would require a very large number of relays (approximately half the number of hosted onion services at a given time point) acting as HSDir. Since the Tor specification limits the number of relays per IP address to two, it would also require a high number of unique IP addresses apart from the systems. To be able to collect the descriptors more efficiently, they use so-called *shadow relays*. These are relays that do not appear in the consensus but are still monitored by the directory authorities and thus obtain flags. If one of the two running relays under one IP address fails, a shadow relay takes over, which then also acts as HSDir, since it has already received the required flags. This way it is possible to fade in and fade out HSDirs. Using this method, it was possible to operate 600 Tor relays on just 50 IP addresses. Under each IP address 24 relays were operated, which were gradually disabled over a period of 24 hours to cover the complete range of the Tor *distributed hash table* (DHT). The described attack is now no longer possible because the flag assignment vulnerability has been fixed and the distribution of onion services V3 on the DHT works in a different way such that positions can not be pre-calculated.

Owen et al. [19] similarly to Biryukov et al. [18], collect the addresses of onion services by placing nodes in the DHT. As the use of shadow relays was no longer possible due to a fix in the Tor source code, the resources in terms of time and relays are considerably higher. They were able to collect approximately 80.000 unique onion addresses, which they subsequently classified. From the results it can be concluded that about 80% of the requests at the time were related to onion services that offered abusive sites (sites where the title suggests some form of sexual abuse).

Elices et al. [20] propose a method for the identification of a hidden server under the assumption that the adversary operates on an ISP level and is able to monitor the traffic between the onion server and its guard node. With access to this information, they are able to correlate the increment of TCP traffic on the link between the hidden server and its guard node with a client request and thus get the IP address of the hidden server.

Biryukov [18] et al. and Ling et al. [21] both propose a method to determine whether a relay controlled by the attacker is the guard-node of an onion service based on characteristics of the Tor protocol. These attacks offer a high accuracy in the identification of guard nodes, but are limited to one onion service per attack. Chen et al [22] resolve this problem by embedding a identifier for each attacked onion service as watermark to distinguish them. According to the authors, this speeds up the attack process by a factor of 11.6.

Current research is focused on the identification of the IP address of onion services and on harvesting onion service addresses. Almost every study contains measures to increase the anonymity of onion services, but metadata is often disregarded. In the course of this work, a protocol was developed that minimizes the linkability of an onion service through its metadata.

3 Background

3.1 Tor

Tor is an open-source *Onion Routing* network that enables bi-directional anonymous communication for TCP-based applications [4]. The goal of this network is to enable users to hide their identity from services they use and limit the ability of traffic analysis [23]. This technique can be used, amongst others, to circumvent censorship measures and access policies, to disguise the web-surfing habit of users or to anonymously provide services to other users. In an ordinary network architecture, the routing information provides details about the identity and location of the requesting user. Even if the payload is encrypted, the routing information has to be transmitted in plain text, as the routers need to be able to read it to find the right path [23].

This issue is addressed by means of Onion Routing. Each user runs a local client referred to as *Onion Proxy* (OP), which handles TCP connections from applications and fetches directory information which consists, amongst others, of a list of known Onion Routers (ORs) and their current state. As Tor utilizes the standardized *SOCKS* proxy protocol as specified in the *RFC1928* [24], most applications can use it without major modifications. The OP selects a path through the network referred to as *circuit*, which consists of OR nodes that only know their direct successor and predecessor. An example of such a circuit is depicted in Figure 3.1.

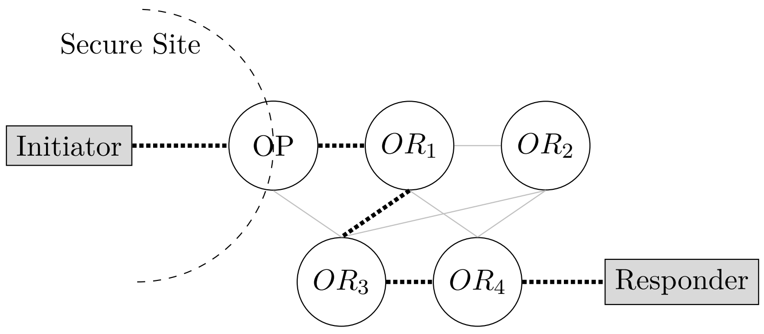


Figure 3.1: Onion Routing Network [23]

In this example the *Initiator* wants to transmit data to the *Responder*. For this purpose its OP builds a circuit through OR_1, OR_3 and OR_4. Each OR holds a long-term identity key, which is mainly used to sign TLS-certificates, router information and directories and a short-term onion key [4]. This onion key consists of a public part, which is made available to all network participants, and a private part, which is only known to the OR. The public part of the onion key is used to encrypt requests during circuit creation, which can only be decrypted by the OR, who is in possession of the corresponding private onion key. Communication in the Tor network is based on fixed-sized *cells* of 512 bytes, each consisting of a header and a payload. The header contains a circuit identifier *circID* and a command, that specifies how the data in the payload is processed. A fundamental distinction is made between *control* cells and *relay* cells. While control cells are interpreted directly by the receiving node, relay cells are used to relay end-to-end data streams. A graphical representation of the structure of those two cell types is given in Figure 3.2. In comparison to control cells, relay cells have an additional header containing a stream identifier (*StreamID*), which allows the multiplexation of multiple streams over a circuit, an integrity checksum, the payload length and a relay command. This relay command is utilized for the management of data streams over the circuit. [4]

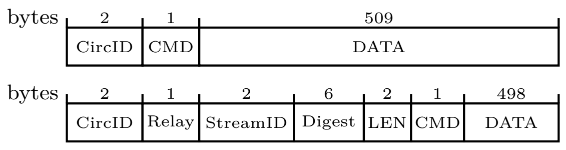


Figure 3.2: Tor Cell Types [4]

The subsequent information on the Tor protocol procedure is taken from the Tor specification [25]. In order to create a new circuit, the initiator's OP needs to send a control cell containing the *create* command to the first hop in the chosen path. For ease of reference, the path from the example in Figure 3.1 is used. The OP chooses a circID not currently in use and sends the first half of its handshake to OR_1. This handshake is necessary for the establishment of a shared symmetric key between the OP and each OR on the path. Currently, two different types of handshakes are supported, which are both based on the *Diffie-Hellman* (DH) key exchange:

- **The *Tor Authentication Protocol* (TAP) Handshake**

  Within the course of this handshake option, a regular DH handshake in $\mathbb{Z}_p$ is performed. For this purpose, the initiator generates a private key x randomly. The initiator constructs an onion-skin, which refers to the DH public key g^x. This onion-skin is encrypted using the *legacy hybrid encryption* algorithm, in the course of which an additional symmetric key K is generated, that is then encrypted together with the first part of the public DH key (g^x) and additional padding with the public short-term onion key of the second participant using *RSA*. The second part of the public DH key (g^x) is encrypted using the symmetric key K. Both encrypted parts are transmitted to the second participant as the payload of a create cell.

  The second participant first decrypts the ciphertext containing the symmetric key and the first half of the DH handshake using its private key. Using the decrypted symmetric key K, it can now decrypt the second part of the DH handshake and can obtain g^x. After receiving the DH public key, the participant randomly generates a private key y and is now able to compute the shares key g^{xy}. It sends its DH public key g^y along with data that demonstrates knowledge of the computed shared key to the initiator. Once the handshake is completed, the initiator can calculate the shared key g^{xy}. This shared key is used as an input to a *Key Derivation Function* (KDF) that outputs keys to secure the connection between the two participants.

- **The ntor Handshake**

  With this handshake option, multiple DH handshakes are used to generate a shared key. The following calculations are carried out on the group *curve25519* as specified by Bernstein [26].

In order to perform this handshake, the initiator needs to know the public ntor onion key (hereinafter referred to as B) and the digest of the identity key of the second participant (Hereinafter referred to as ID). In the first step the initiator calculates a temporary keypair x, X using a curve25519 key generation algorithm and generates a client-side handshake containing the identity digest of the other participant, its ntor onion key and the public part of the newly generated key pair (X). This handshake is referred to as client handshake and is sent to the other participant, which in turn generates a keypair y, Y and computes the following values using its private ntor onion key b:

$$\begin{aligned} secret\_input &= EXP(X,y) \mid + EXP(X,b) \mid ID \mid B \mid X \mid Y \mid PROTOID \\ KEY\_SEED &= H(secret\_input,\ t\_key) \\ verify &= H(secret\_input,\ t\_verify) \\ auth\_input &= verify \mid ID \mid B \mid Y \mid X \mid PROTOID \mid "Server" \end{aligned} \tag{3.1}$$

The values *PROTOID*, *t_key* and *t_verify* are statically defined strings. The participant generates its handshake consisting of the public key of the newly generated key pair (Y) and a hash of *auth_input* and sends it to the initiator. This is referred to as the server handshake. The initiator checks whether the received public key is a group member and calculates:

$$\begin{aligned} secret\_input &= EXP(Y,x) \mid EXP(B,x) \mid ID \mid B \mid X \mid Y \mid PROTOID \\ KEY\_SEED &= H(secret\_input,\ t\_key) \\ verify &= H(secret\_input,\ t\_verify) \\ auth\_input &= verify \mid ID \mid B \mid Y \mid X \mid PROTOID \mid "Server" \end{aligned} \tag{3.2}$$

The initiator can now generate a hash of *auth_input* and compare this value with the value it received from the other participant. If these match, both parties are now in possession of a key seed, which serves as input for a KDF that generates the keys needed to secure the communication.

As the TAP handshake is rather slow due to the usage of RSA, the ntor handshake is preferred. After completing the handshake, relay cells between OP and OR_1 are encrypted using the established key. To extend the circuit, the OP sends a relay cell with the command extend to OR_1, which contains the next hop on the path and the ntor client handshake with OR_3. OR_1 creates a circuit to OR_3 and decrypts the relay cell it received from OP. It copies the received half client handshake into a create cell and passed this cell to OR_3. OR_3 completes the handshake and responds with a control cell with the *created* command, which includes the server handshake. OR_1 receives the response and forwards it to OP using a relay cell. To extend the circuit, the OP acts as described above, always asking the current last node in the path to extend one hop further. Once the circuit is established, the OP shares a session key with each node on the path.To transmit data anonymously, the OP encrypts the data layer by layer with the established session keys of the respective OR starting with the last hop on the route [23]. This layered encryption scheme inspired the name of this mechanism, as it is reminiscent of the layers of an onion, as illustrated in 3.3. The ellipses show the encryption layers of the onion using the session keys of the ORs (PK_{OR_x}). On the way from the OP to the responder, each node removes a layer of encryption so that the last node in the path can read the plaintext and make a request to the desired resource. The responder's reply to the OP is secured in the same way. Each OR on the path adds a layer of encryption using its shared key with the OP, such that only the OP can decrypt it.

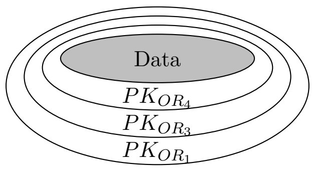


Figure 3.3: Onion Structure

3.2 Onion Services

Onion services, formerly known as hidden services, offer a method for providing bidirectional TCP-based services on the internet while maintaining the anonymity of the provider. This is achieved through the use of an intermediary node through which the communication is handled. Both the client and the service use the Tor network to establish an anonymous connection to this intermediary, thus allowing communication to take place without the client knowing the location of the service and the service knowing the location of the user. As the IP address specifies the location of a network device, the service is now no longer contacted via its IP address, but instead via an address that is specific to the Tor network. This address is referred to as *onion address* and is unambiguously associated with the service. Despite the need to install specialized software for the set-up of such onion services, no changes to the service to be provided are necessary.

The behaviour and functioning of onion services are specified in the *Tor Rendezvous Specification*, which is currently available in version three [25] and is the basis for the information provided in this chapter. The following subsections describe the content of this specification and the process for providing and accessing onion service in detail. Figure 3.4 provides an illustrated overview of the steps involved in this process. The numbered labels match the respective textual descriptions in the subsections. [4, 25]

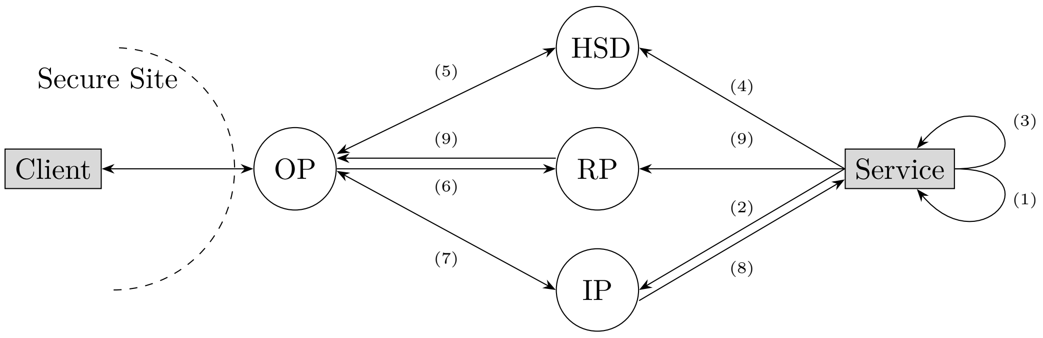


Figure 3.4: Onion Service Architecture

3.2.1 Service Initialization

1. **Generate Long-term Identity Key**

   Initially, the provider generates a long-term key pair, which identifies the service. This key pair is kept offline and is solely used for the generation of blinded signing keys. A detailed description of the menagerie of keys used and their interrelationships is provided in section 3.2.3.

2. **Choose Introduction Points**

   Once the identity keypair is created, the hidden service selects several Tor nodes as its introduction points.These introduction points represent the initial contact points for clients to contact an onion service. The number of introduction points a service uses ranges between zero and twenty. If an onion service has zero introduction points, it cannot be reached by any client. The default number of introduction points defined in the specification is three [25]. As first part of the selection process, the onion service builds an anonymous three-hop circuit to its randomly chosen introduction points. In case of a single onion service being used, which trades service-side location privacy for improved performance [27], non-anonymous one-hop circuits are utilized for that purpose.

   Subsequently, the onion service generates a short-term introduction point authentication key pair, which is used to ensure the authenticity of messages passed between the onion service and the introduction point during the establishment of the introduction point. Furthermore, the public component of this key is included in requests from clients to introduction points to ensure that the intended introduction point was reached. In order to register an introduction point, the onion service first builds a circuit to that node and sends an *ESTABLISH_INTRO* relay cell, which contains the public part of the generated introduction point authentication key, the length and type of the introduction point authentication key, possible extension (e.g. DoS defence), a *Message Authentication Code* (MAC) and a signature, which is signed with the private part of the introduction point authentication key. When a Tor node receives an *ESTABLISH_INTRO* cell, it checks whether:

   - The key type is recognized
   - The key is well formatted
   - The signature is correct (using the public key included in the request)
   - The MAC is valid
   - The circuit isn't already a rendezvous circuit or an introduction circuit
   - The key isn't already in use

   If any of this checks fail, the Tor node rejects the cell and tears down the circuit. If all above verifications pass, the introduction point reports its status back to the onion service using an *INTRO_ESTABLISHED* relay cell. From this point on, the Tor node acts as an introduction point for the onion service.

3. **Build Service Descriptors**

   In the next step the onion service generates a set of descriptors, which contain, among other things, information on the introduction points of the onion service, which is necessary in order to contact them. As these descriptors are going to be encrypted twice, the first structure generated by an onion service is referred to as the second layer plain text. The format of this second layer plaintext is depicted in Figure 3.5.

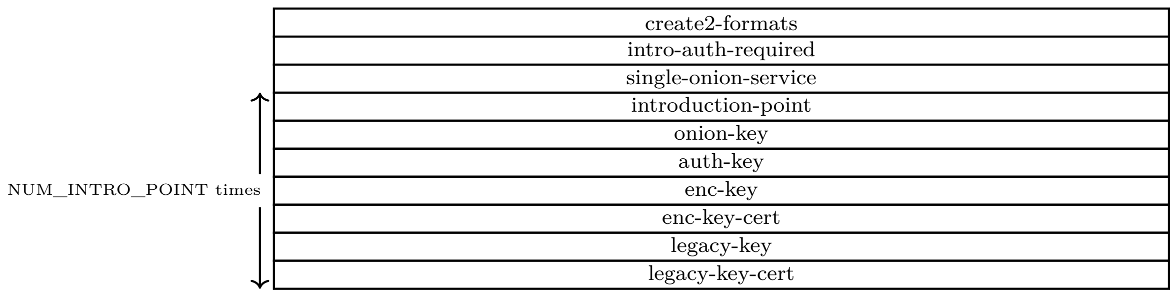


Figure 3.5: Onion Service Descriptor Second Layer Plaintext Format

The field *create2-formats* contains a list of accepted *CREATE2* cell types. Thereby, at least the current default handshake type *ntor* must be supported. As an onion service may restrict access to authorized users, the client must be aware of the introduction-layer authentication types the onion service supports. These types are stored in the field *intro-auth-required.* Currently, authentication is possible by the use of an *Ed25519* key pair or a password. The field *single-onion-service* indicates whether the onion service is provisioned as a single-onion service using non-anonymous circuit connections to introduction points and rendezvous points. After this block specific to the onion service the second-layer-plaintext contains the following items for each of its introduction points.

- *introduction-point*: Base64 encoded link specifiers and identities of the Tor node acting as introduction point
- *onion-key*: Base64 encoded medium-term ntor onion key of the introduction point used when a client extends to it.
- *auth-key*: A certificate cross-certifying the public introduction point authentication key using the descriptor signing key.
- *enc-key*: Base64 encoded public key used to encrypt requests to the onion service.
- *enc-key-cert*: A certificate cross-certifying the public encryption key using the descriptor signing key.
- *legacy-key*: RSA public key used for a legacy introduction point.
- *legacy-cert*: A certificate cross-certifying the legacy key using the descriptor signing key.

The second-layer plaintext is encrypted as specified in equation 3.3, whose design is intended to protect the confidentiality of the descriptor against unauthorized clients.

$$ENCRYPTED = STREAM(SECRET\_IV,\ SECRET\_KEY)\ XOR\ PLAINTEXT \tag{3.3}$$

The initialization vector *SECRET_IV* and the key *SECRET_KEY* used for this encryption are computed as denoted in equation 3.16 in section 3.2.3.6. If client authentication is enabled, the onion service generates 32 random bytes as a *descriptor_cookie*. In case client authentication is not used, the *descriptor_cookie* is left blank. The following parameters are used for the key derivation:

- SECRET_DATA = blinded-public-key | descriptor_cookie
- STRING_CONSTANT = "hsdir-encrypted-data"

The ciphertext is stored in the *encrypted* field of the first-layer plaintext, whose structure is depicted in Figure 3.6.

desc-auth-type
desc-auth-ephermeral-key
auth-client
encrypted

Figure 3.6: Onion Service Descriptor First Layer Plaintext Format

The field *desc-auth-type* describes the type of authorization which is used to protect the descriptor. Currently, *"x25519"* is the only valid value for this field, which represents the multi-staged cookie encryption scheme described below . For this the onion service generates an ephemeral x25519 key pair, whose public key is stored in the *desc-auth-emphemeral-key*. It is assumed that each authorised client holds a shared x25519 public key with the onion service. If client authentication is enabled, the descriptor_cookie is encrypted for each authorized clients. As multiple clients could be authorized for one onion service, the *auth-client* section contains one encrypted descriptor_cookie per client. The keys for this encryption are computed as follows:

$$\begin{aligned} SECRET\_SEED &= x25519(hs\_y,\ client\_X) \\ KEYS &= KDF(subcredential \mid SECRET\_SEED, 40) \\ CLIENT-ID &= \text{fist 8 bytes of KEYS} \\ COOKIE-KEY &= \text{last 32 bytes of KEYS} \end{aligned} \tag{3.4}$$

where *hs_y* denotes the associated ephemeral *x25519* privat key for the public key in *desc-auth-emphemeral-key*, *client_X* is referred to as the pre-shared *x25519* public key of the authorized client and *subcredential* is generated during the key blinding phase for each period and can be generated by clients with the knowledge of the onion service public identity key.

Each line in *auth-clients* contains the base64 encoded *CLIENT_ID*, a base64 encoded initialization vector *iv* and the base64 encoded encrypted cookie in the *encrypted-cookie* field. The cookie is encrypted as described in equation 3.5.

$$encrypted-cookie \;=\; STREAM(iv,\; COOKIE-KEY)\; XOR\; descriptor\_cookie \tag{3.5}$$

The first-layer plaintext is eventually encrypted using the same cipher as in equation 3.3. The following parameters are used for the key derivation, based on equation 3.16 in section 3.2.3.6:

- SECRET_DATA = blinded-public-key
- STRING_CONSTANT = "hsdir-superencrypted-data"

This can be seen as a first step towards access control, as a client needs to know the unblinded onion address in order to calculate the blinded onion address for the respective time period. The encrypted data blob is base64 encoded and stored in the field *superencrypted* of the onion service descriptor structure, which is illustrated in Figure 3.7. The format follows the same meta structure as other Tor directory objects [25].

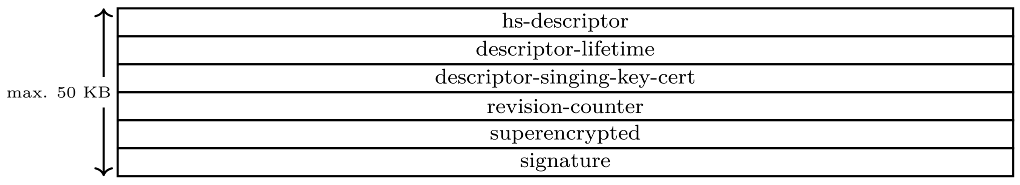


Figure 3.7: Hidden Service Descriptor Format

The field *hs-descriptor* contains the version number of the descriptor, which currently is three. The lifetime of the descriptor is specified in the field *descriptor-lifetime* and ranges between 30 and 720 minutes. A HSDir invalidates a descriptor when its lifetime is exceeded. The field *descriptor-signing-key-cert* contains a certificate that cross-certifies the short-term descriptor signing key with the blinded public key. The format of the certificate differs from standardized certificate formats and is specified in the *tor proposal 220* concerning the migration from *RSA* to *Ed25519* keys [28]. Its structure is depicted in Figure 3.8.

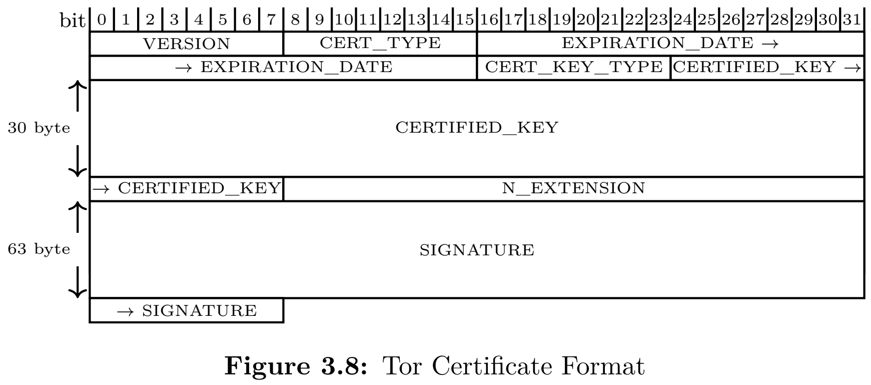


Figure 3.8: Tor Certificate Format

If an HSDir receives multiple descriptors for a descriptor-singing-key, the descriptor with the higher revision number specified in the *revision-counter* field is sent to the requester. The field *superencrypted* contains the twice encrypted and *base64* encoded blob, whose creation is described in detail above.

4. **Publish Service Descriptors**

   In order to publish and fetch onion service descriptors, both services and clients must be able to examine the directories responsible for the service in the given time period. This is accomplished by the usage of *HSDir hash rings*, whose structure is depicted in Figure 3.9.

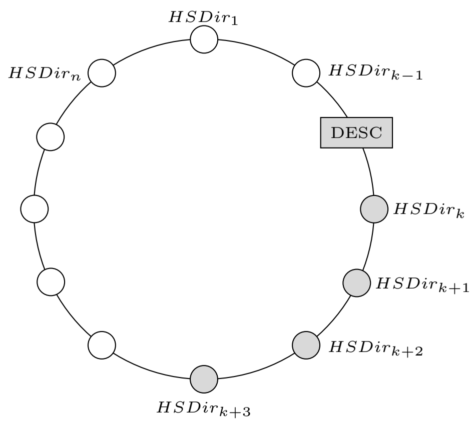


Figure 3.9: Onion Service Hash Ring Structure

For each HSDir in the consensus a hash value is calculated that includes information about the identity of the directory, as well as information about the current time period. This hash value specifies the position of the directory in the hash ring, as indicated by $HSDir_x$ in the figure above, where x ranges from one to n depending on the position of the descriptor in the hash ring. This position is computed on the basis of the blinded public key of the service and information of the current time period and is denoted as $DESC$ in Figure 3.9. By default, the descriptor is uploaded to the first four directories ($HSDir_k$ to $HSDir_{k+3}$), which numerically directly follow the position of the descriptor in the hash ring. A detailed description of the calculation of indices is provided later in this chapter.

To prevent a set of descriptor directories from censoring a service, the onion service uploads the descriptor to several different directories that vary over time. The frequency of directory changes (i.e. the length of the time period) depends on the *hsdir-interval* parameter, which is a consensus parameter established between directories. The current default value for *hsdir-interval* is 1140 minutes, which equals to one day. The time period number, which is necessary for the calculation of the index of the onion service descriptors and its responsible directories, is determined as shown in equation 3.6.

$$period\_num = \frac{minutes\_since\_epoch - rotation\_time\_offset}{time\_period\_length} \tag{3.6}$$

The parameter *minutes_since_epoch* stands for minutes since the Unix epoch[1]. As the developers of *hidden services* wanted to start the time period at *12:00 UTC*, an offset is deducted (*rotation_time_offset*). Finally, the difference is divided by the period length, which is specified in *hsdir-interval*. The result of the calculation is the time period number (*period_num*). To calculate the position of a service descriptor within the hash ring, the following parameters from the *consensus* are required:

- *hsdir_n_replica*: Number of replicas used for uploading the onion service descriptor in the range of [1,16] with a default value of 2.
- *hsdir_spread_fetch*: Number of nodes a client chooses from, when fetching a onion service descriptor in the range of [1,128] with a default value of 3.
- *hsdir_spread_store*: Number of nodes to which an onion service descriptor is uploaded in the range of [1,128] with a default value of 4.
- *valid-after*: Uniform time value for calculating the current period number to minimize the desynchronization of clients and services by asynchronous clocks.
- *router_status*: List of onion router nodes including their identity, address, port and flags.

Now for every time period, the onion service calculates:

$$\begin{aligned}
&\text{for replicanum in 1...hsdir\_n\_replicas:}\\
&hs\_index(replicanum) \;=\; H(\text{"store-at-idx"} \mid\\
&blinded\_public\_key \mid\\
&INT\_8(replicanum) \mid\\
&INT\_8(period\_length) \mid\\
&INT\_8(period\_num))
\end{aligned} \tag{3.7}$$

The parameter *replicanum* is in the range of *hsdir_n_replica*. The *blinded_public_key* is derived from the public master identity key as described in section 3.2.3.2. The resulting *hs_index* is the index of the descriptor in the hash ring. In the next step, for every entry in the *router_status* consensus parameter that has the *HSDirV3* flag set, the directory index is calculated as follows:

$$\begin{aligned}
&hsdir\_index(node) \;=\; H(\text{"node-idx"} \mid node\_identity \mid\\
&shared\_random\_value \mid\\
&INT\_8(period\_num) \mid\\
&INT\_8(period\_length))
\end{aligned} \tag{3.8}$$

As for the calculation of the *period_num* in equation 3.7, *period_length* is taken from the consensus. To limit the predictability of the position of a onion service descriptor in the

[1] 1970-01-01 00:00 UTC

hash ring, the authorities generate an additional random value (*shared_random_value*(srv)), which is added to the consensus and used in equation 3.8. As the transition of time period and random values does not happen simultaneously, at every point in time there are two valid random values. A detailed description of this behaviour can be found in chapter 3.2.4. When an onion service wants to publish its descriptors, it calculates *hs_index* for every replica and *hsdir_index* for every directory in the consensus. The service uploads the descriptors to the first *hsdir_spread_store* (denoted as $HSDir_k$ to $HSDir_{k+3}$ in Figure 3.9) nodes that immediately follow the numerical value of *hs_index*. The actual upload is performed using an *HTTP POST* request to the selected directory.

3.2.2 Service Access

5. **Fetch Service Descriptor**

   In order to retrieve the descriptor from the responsible HSDirs, the client requires the onion address of the services, which must be transmitted using any kind of out-of-the-band communication. Once the client is in possession of the onion address, it is used to calculate the blinded onion address for the current time period, as described in 3.2.3. For each node in the consensus that obtained the HSDir (V3) flag, the *hsdir_index* is calculated as described in 3.2.1. Subsequently, the *hs_index* for both replicas is computed, based on the blinded public key and the current time period. Now the client randomly chooses one directory randomly from the set of responsible directories, which is composed of the first *hsdir_spread_fetch* directories which numerically follow the *hs_index*. To fetch the descriptor, a *HTTP GET* request for the URL */tor/hs/<version>/<z>* is initiated, where *<z>* represents the base64 encoded blinded public key of the onion service and version stands for the onion service protocol version. If the connection attempt fails, or the descriptor is not found by the directory, other directories form the previously chosen subset are queried.

6. **Establish Rendezvous Point**

   The client decrypts the descriptor and now knows the introduction points of the service, which can be considered as initial contact points. Prior to connecting to the introduction point of an onion service, the client must establish a connection to a rendezvous point, which acts as an intermediary between client and service. For this purpose, the client selects a random Tor node, establishes a circuit to that node and sends an *ESTABLISH_RENDEZVOUS* cell containing a *RENDEZVOUS_COOKIE*. This cookie is an arbitrary 20-byte value and is later used when the onion service connects to the rendezvous point. The rendezvous points acknowledges the establishment of the rendezvous point by responding to the client with a *RENDEZVOUS_ESTABLISHED* cell.

7. **Connect to Introduction Point**

   To contact the service, the client builds an anonymous circuit to one of the introduction points contained in the descriptor. Then it sends a *INTRODUCE1* cell with the following content to the introduction point:

- *LEGACY_KEY_ID*: Used to differentiate between legacy and new style introduction cells. If new style cells are used, this field is zeroed.
- *AUTH_KEY_TYPE*: Type of the authentication key used (currently only the authentication using an *Ed25519* public key is possible).
- *AUTH_KEY_LEN*: Length of the authentication key.
- *AUTH_KEY*: Introduction point authentication key (taken from the second-layer plaintext of the descriptor)
- *EXTENSIONS*: Possible extensions.
- *ENCRYPTED*: Encrypted plaintext, which provides information about the RP selected by the client. Dependent on the type of encryption key used, there are multiple ways of decrypting this ciphertext.

In addition to exchanging information about the RP of the client, handshake data is exchanged using the *INTRODUCE1* cell. Thereby, the following requirements are specified for a handshake as part of the introduction protocol:

- The onion service must be able to decrypt additional information, that contains the rendezvous token and the information required to extend to this RP.
- During the handshake, a set of shared keys is established, which can later be used to secure traffic between onion service and client.
- The onion service must be able to verify the authenticity of the cell, such that any modification or manipulation since the creation of the cell would be noticed.

The Tor Authentication Protocol (TAP) handshake, which is now considered obsolete, would meet the first two requirements, but since it does not make use of a MAC, the last requirement would not be fulfilled. An example for a handshake that satisfies all requirements is the *ntor* handshake, whose description is provided in section 3.1. In the context of onion service, several modifications of the handshake protocol are necessary. The modified protocol is shown in equation 3.9.

$$\begin{aligned} secret\_input &= EXP(B,x) \mid AUTH\_KEY \mid X \mid B \mid PROTOID \\ info &= m\_hsexpand \mid subcredential \\ hs\_keys &= KDF(intro\_secret\_hs\_input \mid t\_hsenc \mid \\ &\qquad info,\ S\_KEY\_LEN + MAC\_LEN) \\ ENC\_KEY &= hs\_keys[0 : S\_KEY\_LEN] \\ MAC\_KEY &= hs\_keys[S\_KEY\_LEN : S\_KEY\_LEN + MAC\_KEY\_LEN] \end{aligned} \tag{3.9}$$

Initially, the client generates a curve25519 keypair consisting of the private key x and the public key X. The ntor onion key referred to as B in the regular ntor handshake is replaced by the public introduction point encryption key. Furthermore, the node id is substituted by the *AUTH_KEY* generated previously by the onion service for the specific introduction point. The fields *t_hsenc*, *t_hsverify*, *t_hsmac* and *m_hsexpand* are text constants which are not discussed further. As the encrypted part of the *INTRODUCE1* cell, the client sends:

- CLIENT_PK
- ENCRYPTED_DATA
- MAC

where the *CLIENT_PK* is the public key X, *ENCRYPTED_DATA* corresponds to the original structure of the encrypted field, which contains information about the RP and is encrypted with the symmetric *ENC_KEY*. The *MAC* field contains an MAC of the entire cell using the computed *MAC_KEY*.

8. **Relay Introduction Cell**

   The introduction point checks whether the authentication key matches an auth key of an active introduction circuit. If that is the case, it sends an *INTRODUCE2* cell with exactly the same content as the *INTRODUCE1* cell to the os and notifies the client of the successful referral with an *INTRODUCE_ACK* cell.

9. **Join Rendezvous Point**

   The onion service receives the *INTRODUCE2* cell and checks if the *AUTH_KEY* or the *LEGACY_KEY_ID* matches the keys for this introduction circuit. If the key match, the next step is to verify whether the onion service has already received a cell with the same content. If this is the case, the cell was replayed and is silently dropped by the onion service. Otherwise, the service generates a curve25519 keypair y, Y, where y denotes the private key and Y represent the public key. The onion service computes:

$$\begin{aligned} secret\_input &= EXP(X, b) \mid AUTH\_KEY \mid X \mid B \mid PROTOID \\ info &= m\_hsexpand \mid subcredential \\ hs\_keys &= KDF(intro\_secret\_hs\_input \mid t\_hsenc \mid \\ &\qquad info,\ S\_KEY\_LEN + MAC\_LEN) \\ ENC\_KEY &= hs\_keys[0 : S\_KEY\_LEN] \\ MAC\_KEY &= hs\_keys[S\_KEY\_LEN : S\_KEY\_LEN + MAC\_KEY\_LEN] \end{aligned} \tag{3.10}$$

The onion service is now able to decrypt the ciphertext using the *ENC_KEY* and check the authenticity and integrity of the cell using the *MAC_KEY*. The plaintext of the *ENCRYPTED* field is structured as listed below:

- RENDEZVOUS_COOKIE: Rendezvous cookie provided by the client.
- EXTENSIONS: Possible extension.

- ONION_KEY_TYPE: Type of the onion key (either TAP or ntor).
- ONION_KEY_LEN : Length of the onion key.
- ONION_KEY: Onion key that must be used when extending to the rendezvous point.
- NSPEC: Refers to the number of link specifiers included in the cell.
- NSPEC times: LSTYPE, LSLEN, LSPEC: Entry for each link specifier provided including its type and length.
- PAD: Optional padding .

In the next step the onion service finishes the ntor handshake by calculating:

$$\begin{aligned}
rend\_secret\_hs\_input &= EXP(X,y) \mid EXP(X,b) \mid AUTH\_KEY \mid B \mid \\
&\quad X \mid Y \mid PROTOID \\
NTOR\_KEY\_SEED &= MAC(rend\_secret\_hs\_input,\ t\_hsenc) \\
verify &= MAC(rend\_secret\_hs\_input,\ t\_hsverify) \\
auth\_input &= verify \mid AUTH\_KEY \mid B \mid Y \mid X \mid PROTOID \mid "Server" \\
AUTH\_INPUT\_MAC &= MAC(auth\_input,\ t\_hsmac)
\end{aligned} \tag{3.11}$$

As mentioned for the client ntor handshake, *t_hsenc*, *t_hsverify*,*t_hsmac* and *m_hsexpand* are text constants, which are not described further. The onion service generates a handshake reply that contains its generated public key (Y) and the *AUTH_INPUT_MAC*. Subsequently the onion service builds a circuit to the RP and sends a *REDENZVOUS1* cell, which consists of the handshake reply in the form of *HANDSHAKE_INFO* and the rendezvous cookie. If this cookie corresponds to a rendezvous cookie assigned to a circuit that is not yet connected, the RP connects the two circuits and sends a *RENDEZVOUS2* cell to the client containing the *HANDSHAKE_INFO*. The client is now able to finish the ntor handshake as follows:

$$\begin{aligned}
rend\_secret\_hs\_input &= EXP(Y,x) \mid EXP(B,x) \mid AUTH\_KEY \mid B \mid \\
&\quad X \mid Y \mid PROTOID \\
NTOR\_KEY\_SEED &= MAC(ntor\_secret\_input,\ t\_hsenc) \\
verify &= MAC(ntor\_secret\_input,\ t\_hsverify) \\
auth\_input &= verify \mid AUTH\_KEY \mid B \mid Y \mid X \mid PROTOID \mid "Server" \\
AUTH\_INPUT\_MAC &= MAC(auth\_input,\ t\_hsmac)
\end{aligned} \tag{3.12}$$

After the validation of the authenticity of the received data by comparing AUTH_INPUT_MAC to the *AUTH* field received in the *HANDSHAKE_INFO*, both the client and the onion service are in possession of handshake output to derive shared keys that are used encrypt and authenticate data end-to-end between them.

3.2.3 Key Types

To ensure the integrity and authenticity of onion services, a variety of different keys are used, whose hierarchy is shown in Figure 3.10. The only key needed to identify and access an onion service is the public master identity key and its derived blinded public key. The remaining keys are used internally for authentication and encryption. The following section describes the use and origin of each key species.

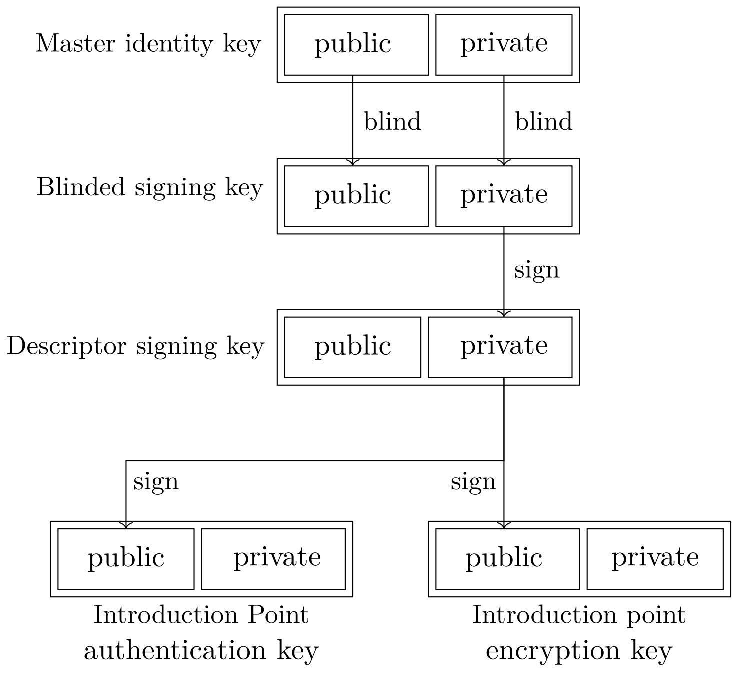


Figure 3.10: Onion Service Key Hierarchy

3.2.3.1 Master Identity Key

The long-term identity key-pair is used for the generation of the blinded signing keys and can be kept offline. As proposed in the Tor specification [29], the *Ed25519* key signature system based on the elliptic curve *Curve25519* is used.

3.2.3.2 Blinded Signing Key

For every time period the onion service uses a different blinded signing key to sign directory information. Given a nonce *n*, from the keypair (signing_key, public_key) a new keypair (signing_key_n, public_key_n) can be derived. Without the knowledge of *public_key*, the key *public_key_n* cannot be derived. Furthermore it is possible to check a signature signed with *signing_key_n* with *public_key_n*. The complete procedure for blinding a key works as follows:

Given is a master identity key pair (a,A), where a denotes the private key, A refers to the public key and B is the base point of the elliptic curve as specified in [30], such that $A = aB$ and l is the prime order of *B*. To derive a blinded key pair (*a'*, *A'*) for a nonce *N* and an optional secret *s*, several blinding factors are calculated:

$$\begin{aligned} BLIND\_STRING &= \text{"Derive temporary signing key"} \mid INT\_1(0) \\ N &= \text{"}key-blind\text{"} \mid INT\_8(period-number) \mid INT\_8(period\_length) \\ B &= Ed25519 \text{ base point as specified in [30]} \\ h &= (BLIND\_STRING \mid A \mid s \mid B \mid N) \end{aligned} \tag{3.13}$$

Additionally, according to the specification of *Ed25519*, some factors of *h* are clamped to specific values. Using the calculated blinding factors, the blinded private key *a'* for a period is calculated as:

$$\begin{aligned} a' &= h\ a\ mod\ l \\ RH' &= SHA-512(RH\_BLIND\_STRING \mid RH)[:32] \\ RH\_BLIND\_STRING &= \text{"Derive temporary signing key hash input"} \end{aligned} \tag{3.14}$$

The blinded public key *A'* is calculated based on the blinded private key as follows:

$$A' = h\ A = (ha)B \tag{3.15}$$

3.2.3.3 Descriptor Signing Key

The descriptor signing key is used to sign the onion service descriptors. In contrast to blinded signing keys, these keys must be stored online at the onion service host. The key itself is signed using the blinded signing key. The descriptor signing key pair is a newly generated *X25519* key pair and is not dependent on any other key used in this cryptographic scheme.

3.2.3.4 Introduction Point Authentication Key

The onion service generates a unique *X25519* key pair for each introduction point it establishes. When a client connects to an introduction point in order to contact the onion service, the introduction point verifies that it maintains a circuit to the service corresponding with that introduction point authentication key. This allows the client to ensure that it is connected to the correct introduction point and the onion service to check whether the contact of the client took place through a known introduction point. The public part of the key is included in the singing-key extension of the onion service descriptors and is cross-certified by the descriptor signing key.

3.2.3.5 Introduction Point Encryption Key

The introduction point encryption key is used to encrypt information about the rendezvous point chosen by the client in the introduction protocol. Furthermore, the key is used as the public key of the onion service in case the advanced ntor handshake is used to exchange shared keys material. The *X25519* key pair is generated by the onion service for every introduction point and is cross-certified by the descriptor signing key.

3.2.3.6 Descriptor Encryption Keys

The key for encrypt hidden service descriptors is derived from secret data provided by the $SECRET\_DATA$ parameter and a string constant given by the parameter $STRING\_CONSTANT$ as follows:

$$\begin{aligned}
SALT &= \text{16 bytes from } H(random)\\
secret\_input &= SECRET\_DATA \mid subcredential \mid INT\_8(revision\_counter)\\
keys &= KDF(secret\_input \mid salt \mid STRING\_CONSTANT,\ S\_KEY\_LEN\\
&\quad + S\_IV\_LEN + MAC\_KEY\_LEN) \qquad (3.16)\\
SECRET\_KEY &= \text{first } S\_KEY\_LEN \text{ bytes of keys}\\
SECRET\_IV &= \text{next } S\_IV\_LEN \text{ bytes of keys}\\
MAC\_KEY &= \text{last } MAC\_KEY\_LEN \text{ bytes of keys}
\end{aligned}$$

3.2.4 Time Periods and Shared Random Values

As mentioned in 3.2.1, onion services maintain two descriptors at any time to be reachable by clients that use the current consensus or the previous one. The descriptors are periodically published to the responsible HSDirs. After each publication of a descriptor, a random timer between 60 and 120 minutes is set. When the timer expires, the service uploads its descriptors again to the HSDirs responsible for that *Time Period* (TP) and *Shared Random Value* (SRV). Each consensus contains the SRV for the current and the previous TP. The procedure for choosing the right SRV along with the TP differs between client and service. The differences can best be explained by using an example, which is illustrated in Figure 3.11. $TP\#X$ indicates, that from this point on a new TP X begins. The symbol \$ indicates a descriptor rotation. $SRV\#X$ means that a new SRV X will be available from that point.

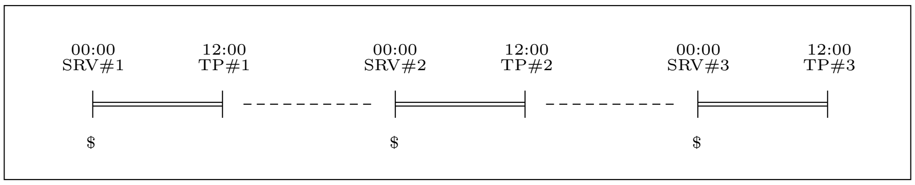


Figure 3.11: Tor Hidden Service Key Hierarchy [25]

The two descriptors uploaded by the service are referred to as *first* and *second* service descriptors and follow different upload logics and purposes:

- **First Descriptor**:

  The first descriptor is intended for clients that have a consensus that is still in the previous TP. To create this descriptor, the previous TP and SRV is used when the service is in the time segment between a new TP and a new SRV (marked with '-' in Figure 3.11). For example, if the service is at 14:00, shortly after $TP\#1$, it would still upload the descriptor with $TP\#0$ and $SRV\#0$. If the service is in the time segment between a new SRV and a new time period (marked with '=' in Figure 3.11) it uses the current TP and the previous SRV. For instance, if the service is at 02:00, shortly after $SRV\#2$, it would upload the descriptor with $TP\#1$ and $SRV\#1$.

- **Second Descriptor**:

  The second descriptor is intended for clients who have an up-to-date consensus in the same TP as the service. In order to create this descriptor, the current TP and the current SRV are always used to upload the descriptor if the service is in the time segment between a new TP and a new SRV (marked with '-' in Figure 3.11). For example, if the service is at 14:00, shortly after $TP\#1$, it would upload the descriptor with $TP\#1$ and $SRV\#1$. If the service is in the time segment between a new SRV and a new TP (marked with '=' in Figure 3.11) it uses the next time period and the current SRV. For instance, if the service is at 02:00, shortly after $SRV\#2$, it would upload the descriptor with $TP\#2$ and $SRV\#2$.

If services receive a consensus with a $valid\_after$ time past the next SRV calculation time, they rotate their descriptors by discarding the first descriptors, pushing the second descriptors to the first descriptors and rebuilding the second descriptors. The client aims to synchronize TPs and SRV when fetching descriptors, so it always tries to use $TP\#X$ with $SRV\#X$.

3.2.5 Security Enhancements

Tor in its original implementation was vulnerable to traffic correlation attacks. Within these attacks, adversaries attempt to de-anonymise users by monitoring both the entrance of user traffic to the anonymity network and its exit [31]. In order to do this, the adversary must control both the first node and the last node in a circuit. If it is able to do so, it knows both the client (because it is directly connected to it) and the resource that the client uses. This type of attack on the Tor network was first demonstrated by Oeverlier et al. [16] in the course of the de-anonymisation of onion servers [31]. Based on ideas of Wright et al. [32] they introduced the idea of fixing the first nodes in a circuit to prevent adversaries from gaining control over them. These first nodes are referred to as *entry guards* or *guard nodes*. An illustration of a three-hop circuit utilizing this guard nodes is provided in Figure 3.12.

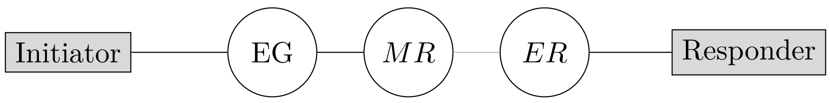


Figure 3.12: Security Enhanced Onion Routing Network [23]

An enhanced circuit consists of the client, in this example referred to as initiator, an entry guard as first hop, a *middle relay* as second hop and an *exit relay* as last hop. The only node that is directly connected to the initiator is the entry guard. The middle relay knows neither initiator nor responder. At the exit relays, traffic exits the Tor network. These are the only nodes that are directly connected to the responder and thus know their location. The entry guards are chosen from a small subset of all relays in the consensus, which are in possession of the *Guard* flag. This flag indicates, that the node is stable, fast and has a specific minimal uptime [33]. This subset is further filtered based on several performance and utility parameters to obtain a set of *primary guards*. Currently, three primary guards are selected, nevertheless it is attempted to use the same guard each time [34]. To keep the probability of success of a traffic correlation attack as low as possible, the entry guard must be persistent. Currently, the lifetime of a guard is 120 days [34]. The middle relay of a circuit is chosen randomly from the relays available in the latest consensus weighted by bandwidth. Similarly, the exit relays are selected, except that their relay must support an appropriate exit policy [34].

Since this makes it very complex to become the guard node of a service, *guard discovery* attacks aim to identify the entry guard of an onion service. Once the entry guard is known to an adversary, an attempt is made to compromise it using other methods. To complicate this, the *Vanguards* extension was developed, which is written in Python. It utilizes the Tor control protocol and consists of three components: [35]

1. **Vanguards**:

   This is the core function of Vanguards. In addition to the entry guards, also middle relays and exit relays are pinned for a longer period of time. With this, the time to discover an entry guard increases from five minutes to weeks or months, even if the attacker is able to control a significant portion of the Tor network. To protect the newly pinned nodes, the rendezvous circuits, directory circuits, and introduction circuits are lengthened. [36]

2. **Bandguards**:

   The Bandguard component checks whether there are traces of bandwidth side channel attacks within circuits, that are used by adversaries to amplify traffic analysis attacks. If such traces are found, an alarm is generated and the affected circuit is optionally closed.

3. **Rendguards**:

   This component tracks the frequency of rendezvous point used on the onion service side. Since some attacks rely on the aid of custom malicious rendezvous point for traffic analysis, an overuse of certain nodes indicates a possible malevolent utilization. If certain threshold values are exceeded, an alarm is generated and the rendezvous circuit is optionally torn down.

4 Unlinkable Onion Services

> "The unlinkability of two or more items of interest from an attacker's perspective means that within the system (comprising these and possibly other items), the attacker cannot sufficiently distinguish whether these items of interest are related or not." [37]

Following this definition, three specific requirements for unlinkable onion services emerge:

1. Recurrent connections to the same onion services must not be linked to a client.
2. Multiple connections of a client to different onion services must not be relatable.
3. Connections of multiple clients to specific onion services must not be associable.

Considering the functional behaviour of onion services in version three, these requirements are only partially met. An adversary controlling one of the Hidden Service Directories (HSDirs) of the onion service cannot identify which users want to access the service, since the Tor clients are connected to the Rendezvous Point (RP) of the service using anonymous circuits, but it can draw conclusions about the number of users and the frequency of access to the service. This can become detrimental for users in case an attacker additionally has access to further sources of information, e.g. the network traffic of the client. Through the correlation of multiple information sources based on the chronological sequence of traffic, the user could be identified and associated with the service. However, for a targeted attack, an adversary would need to know the address of the onion service. With onion services in the previous version two, it was possible to collect addresses on the HSDir, because the required information to determine the address of an onion service was stored unencrypted in the descriptor. This weakness was fixed in onion service version three by securing the descriptor with a multi-layered encryption scheme. Nevertheless, if the onion address is known to the adversary, an analysis of the usage pattern of this service is possible.

Another threat to the unlinkability of onion services is constituted by the monitoring of the host server of the onion service, which typically is a web server or an application server. Theoretically, the usage of an anonymous circuit between the RP and the onion service protects the location of this server, however, it has been shown that revealing the location of the onion service is possible for instance by means of protocol-level attacks [22] [21] [18]. These attacks enable the determination of whether a relay controlled by an adversary is the guard relay of an onion service or the guard relay is adjacent to the controlled relay. If the adversary in fact controlls the guard node, it can monitor the entire traffic between the onion service and its clients and can thus find out how many connections take place and when. Additionally, the adversary would know about the IP address of the onion service. If the guard relay is adjacent to the relay controlled by the adversary, the identity of the guard node is known and the relay can be compromised, surveilled or coerced [35]. Therefore, the Tor project announced these so-called *guard discovery attacks* to be among the most dangerous threats to onion service as of 2018 [38]. The Vanguards addon was developed to address this problem [35], but it is not currently used in the default configuration. Similarly to the monitoring of the HSDir, the adversary needs to know the address of a service to carry out such attacks.

As in both of the above described threats to unlinkability, the identity (i.e. the onion address) of a service plays a key-role, the obvious prevention consists in the decoupling of the onion service and its identity. The following sections provide detailed information on the architecture and efficient implementation of such a decoupled approach.

4.1 Methodological Approach

Uncoupling an onion service from its identity means that the service needs to hand out different custom onion addresses to every client that wants to access it. In order to accomplish that, it is necessary to change the public master key of the service as, beside the version, this key is the only identifier of the service that is encoded in its address [39, 25]. The amendment of the public master key of a onion service can be performed using two different approaches, partly depending on the version of the onion service:

1. **Multiple Onion Services**

   The straight forward way to change the identity of a service is to provision a fresh onion service pointing to the same application. Technically this doesn't correspond to a change of the key, but produces a similar result. The issue with this solution is scalability. In order to provision a new onion service, new keys have to be generated, three additional connections to Introduction Points (IPs) have to be established and maintained and 16 descriptors have to be created and uploaded to the HSDirs. This can best be illustrated by the example of the social media network Facebook, which in 2016 announced, that up to 1.000.000 clients utilized the onion service version of Facebook in a period of 30 days [40]. If each user would be provided with a custom service address, 1.000.000 unique onion services would have to be provisioned. In comparison, the number of unique V2 onion service has never exceeded 240.000 [41].

2. **Multiple Descriptors**

   Rather than provisioning a separate onion service per user, a new descriptor is created for each participant pointing to the same service. The idea of creating custom service descriptors is not new, as it was introduced in the onion service V2 protocol as *stealth* mode for client authorization to hide the activity of the service from unauthorized users. For this purpose, the service generates an asymmetric *client key* and a symmetric *descriptor cookie* per user. After their creation, the service communicates these two values to the authorized user in form of a contact information string. The client key replaces the public master key (referred to as permanent-key in V2), which causes the onion address to change since this is the *base32*-encoded hash value of this key. The descriptor cookie prevents the prediction of the storage location of the descriptor on the HSDir, as this would require the knowledge of the descriptor. Additionally, the descriptor cookie is used to encrypt the IPs of the service, so that only the authorized user can contact the service. This approach is a clear advantage in terms of unlinkability. The onion service now has full control over the dissemination of its descriptors and thus over its accessibility. If the service stops publishing descriptors for a certain user and said user is not able to link descriptors issued for other users to the service, it will no longer get any information on the activities of the onion service. Moreover, the service could theoretically only be found by authorized users, as only they know the onion address of the service. If a non-authorised user nevertheless succeeds in obtaining one of the onion addresses of the service (e.g. by monitoring HSDirs), it cannot use it to contact the service, as the IPs to the service are encrypted. However, this concept does not only have advantages. As the number of descriptors increases, the load on both the onion services, which have to manage and regularly renew these descriptors and the HSDirs, grows. In order to deal with this problem, the maximum number of descriptors an onion service can manage in V2 has been set to 16. [39]

With onion service V3, among other innovation, a novel structure for service descriptors was implemented, which is protected with a double-layered encryption. This prevents adversaries from obtaining onion addresses by monitoring HSDirs. With this new structure and the concept of key blinding, which is described in section 3.2.3.2, the stealth client authorization is no longer needed, since an adversary is not able to find out the onion address of a service, if it is not published. For this reason, the creation of new onion addresses using the client authorization in V3 is no longer possible.

As onion services V2 will no longer be supported in the foreseeable future [15] and onion services V3 don't support stealth client authorization, creating multiple descriptors to generate custom onion addresses and thus provide unlinkability is not feasible at the moment.Therefore, the first prototype implementation of unlinkable onion services will create a custom onion service per user, which implies that the scaling problem still remains. A potential solution could be to not provision a custom service for each user, but instead manage a pool of custom onion addresses and randomly hand out onion addresses from this pool. However, due to time restrictions the solution for this problem is not part of this thesis and is therefore considered future work.

In the next step it must be determined how the generated addresses can be exchanged in the most effective way. In principle, a distinction can be drawn between two approaches, which differ in terms of the size of the expected user pool of a service. These two approaches are described in the following two sections.

4.2 Custom Addressing

Services that have a rather small user pool can take advantage of a simple implementation of unlinkability referred to as custom addressing, whose purpose is the provisioning of a per-client ephemeral onion service and the exchange of the resulting onion address. For the sake of simplicity, the application server, which provides the actual service, and the onion service in combination are referred to as server. The idea of this approach is that the server is aware of all its communication partners due to the limited number of users. The service provisions an onion service for every user it wants to communicate with and sends the custom onion addresses of these services to the respective clients using a secure channel. As no communication from the client to the server is required, only uni-directional communication between server and client is intended. The sequence diagram of the custom addressing procedure is depicted in Figure 4.1.

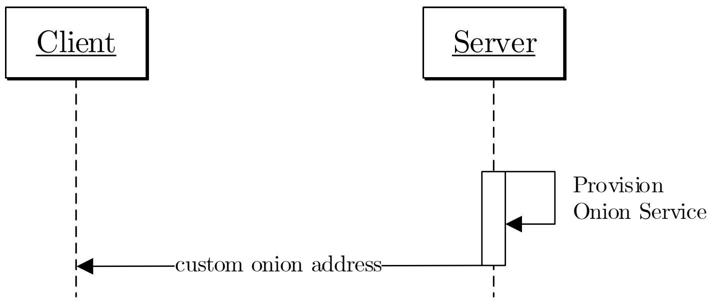


Figure 4.1: Custom Adressing

The major advantage of this approach is its simplicity. No additional complexity is added through the use of a protocol. If the client only uses the provided custom onion address to contact the server, unlinkability is ensured. Furthermore, a server would have full control over its own accessibility since if a server want to be accessed by a client, it simply tears down its custom onion service. In case this client is not able to obtain the custom onion address of another client, it will no longer receive any information about the onion service and its activity. The major disadvantage of this approach is constituted in its scalability. Although there is no theoretical limit of onion services a tor instance can create [25], both the HSDirs and the Tor client are not designed for operating hundreds of thousands of onion services, as for instance the Facebook onion service would require [40].

Furthermore, this approach is based on the assumption that the first contact originates from the server, which excludes any new clients not yet known to the server. This means that a client must be aware of the communication channel for receiving its custom address previous to the first contact. The communication channel cannot utilize the Tor network, as the server has no point of contact with the client, which implies that the server would need to know the identity (e.g. the IP address) of the client. This eliminates the concept of anonymity. An adversary that operates on the Internet Access Provider (ISP) tier would be able to monitor the initial communication and link the client to the server. A remedy could be the use of an anonymous communication service such as *ProtonMail* [42] for the initial communication between the server and the client. ProtonMail uses public-key cryptography to ensure the confidentiality of messages sent between users. For this to work, the server would need to be in possession of a public key of all its users. Due to the restricted user pool, this is feasible. However, the use of communication based on public-key cryptography raises another problem. Both the server and the client must be able to verify the authenticity of the key material used. This could be solved, for example, by using certificates, but this in turn would undermine the principle of anonymity, since the client would need to authenticate itself again a central Certificate Authority (CA) in order to obtain the certificate. To simplify the exchange of custom onion addresses, the *Unlinkable onion Service Protocol* (UOSP) was designed.

4.3 Unlinkable Onion Service Protocol

4.3.1 Concept

As exemplified in section 4.2, the exchange of custom onion addresses between server and client is not a trivial problem. For the sake of usability, accessing an unlinkable service must not differ drastically from accessing a normal onion service from a client's perspective. This implies, that the client must be able to retrieve a custom onion address for a service without additional knowledge and without the need of using a separate secure communication channel. For this purpose, a standardized approach in the form of a protocol must be defined, which specifies the exact procedure from the first request of the client to the receipt of the custom onion address. With the *Unlinkable Onion Service Protocol* (UOSP), such a standardized approach was designed and is described in this section in detail.
The basic conceptual difference of the UOSP compared to *Custom Addressing* consists in the types of addresses a clients uses to contact a service. In a regular onion service access, the service's onion address is used for the initial request and then data is exchanged between client and service. In the UOSP, the address used for exchanging data between client and server is referred to as *data address*. In addition, the UOSP uses a second type of address, whose only purpose is the initial contact of a client to a service and the exchange of each client's custom data onion address. Therefore, this address type is called *introduction address*. Beside their different purposes, the two address types also have different levels of confidentiality. As the introduction address is needed in order to establish an initial contact to a service, this address needs to be publicly available. It could for instance be posted in internet forums or handed out to the clients who should be able to reach a certain service. The data address on the other hand, is personal to a specific client and needs to be kept secret in order to remain unlinkable. If the address were to be published, adversaries could try to manipulate the service's Hidden Service Directories (HSDirs) and thereby gain information about the utilization of the service. This would lead to linkability and render the reason for using the UOSP obsolete.

For this reason, in order to ensure the unlinkability of onion services, the following environmental assumptions are considered to be met:

1. The introduction address of an onion service is published in a way that all clients that need to communicate with the service can obtain it. The address does not have to be kept secret.
2. The data address of an onion service is private to a single client or a client group and is therefore not published in a way that other clients or other client groups that are not authorized can obtain it.

In this context *authorized* means that the onion service has designated the client or the client group to use its data address and is not associated with a cryptographic meaning. Authorization in the cryptographic sense would require that each authorized client obtains information in advance (e.g. a key pair) with which it can unambiguously prove that it is authorized to use a service. However, this would lead to the same implications as with Custom Addressing, that are described in section 4.2.

4.3.2 Architecture

Since each unlinkable onion service needs to provide at least one public introduction address and probable multiple private data addresses, it must provision services for responding to requests to introduction addresses as well as for responding to requests to data addresses. Both of these services must be implemented as onion services, otherwise their location (i.e. the IP address of the service) cannot be hidden. The minimal set of components needed for the provisioning of an unlinkable onion service is illustrated in Figure 4.2.

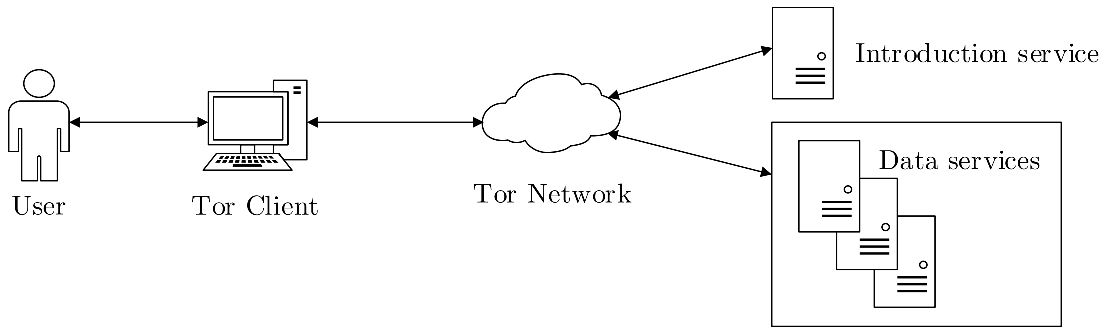


Figure 4.2: Minimal Set of Components Required for Unlinkable Onion Services

The Tor client of the users acts as its onion proxy, which establishes a connection to the Tor network. The introduction service is hosted as a single onion service, whose address represents the UOSP introduction address. Although the introduction service and the data service are illustrated as separate servers, both services must be hosted on a single server because the introduction service must be in control of the provisioning of data services. The application to which the data services point does not necessarily have to be operated on the same server. The basic procedure of a UOSP request is structured as follows:

1. The user obtains the introduction address of an unlinkable onion service through out-of-band-communication (e.g. through an email) or other communication platforms. It is assumed that the creator of the introduction address is trusted.

2. The user then sends a request to the introduction address. The protocol utilized for this request is not limited to web protocols, in principle any TCP-based protocol can be used. Moreover, the environment with which the request is submitted is not specified; it could be a browser, for example. It has to be noted that this request is linkable since a publicly known introduction address of a service is used for this purpose.

3. The request reaches the introduction service, where it is interpreted. At this point, authorization mechanisms such as the client authorization embedded in onion service could be used, which decide whether the service communicates with the user or not. Subsequently, the service provisions a custom onion address and responds to the user.

4. The user checks the well-formedness of the received custom onion address. This can easily be achieved, since the length of a valid onion address is defined in the specification [25]. The obtained custom onion address must be memorized for further requests, as each request would otherwise result in a request to the introduction address, which in turn would lead to linkability.

5. At this point, the user can submit a request to the custom onion address received. The data service responsible for that custom onion address will respond with the requested data.

As indicated in the basic sequence, the user has to perform a number of tasks by himself. This includes the manual query of the custom onion address, the check of its well-formedness and the memorization of previously queried addresses. This might be feasible for technically proficient persons, but is rather unusable for normal users. For this reason, apart from client and server, a third main component in the form of a proxy is introduced. This proxy handles the tasks that previously had to be performed by the user and must be placed between the user and the Tor client, since the data traffic between the Tor client and the Tor network is already encrypted. For reasons of usability, requirements are placed on the transparency and performance of the proxy. If the latency introduced by the proxy is too high, users will not use it [5]. The situation is similar in regard to transparency. If the user experiences a significant increase in the complexity of using a service through the use of the proxy, it will not be used.

4.3.3 Identity Assignment

Essentially, there are two approaches for assigning individual onion addresses to users. These approaches and their advantages and disadvantages are described below.

- **Key Derivation:**

  With this method, a key exchange between proxy and server is initially performed. After the key exchange, both parties are in possession of the same symmetric key, from which further keys can be derived. These derived keys can be used to provision onion services, for which only the client and the server know the address, as its partly based on the symmetric key. To implement this approach, a two-phased protocol was considered in reference to the phase-based mode of operation of Transport Layer Security (TLS) [43]:

  1. Key Exchange Phase:
     This phase serves the establishment of a shared symmetric master key, which could be implemented using an asymmetric key exchange protocol, such as the Diffie-Hellman key exchange. The key exchange protocol requires a number of domain parameters to be agreed upon in advance.
  2. Usage Phase:
     In the second phase, both parties can derive keys from the shared symmetric master key established in phase one. For the key derivation, a double ratchet algorithm is used, which offers future secrecy. The derived key is used to establish an onion service on

the server side, which can only be accessed by the client due to the knowledge of the symmetric key used.

This approach offers the possibility of generating ephemeral onion services, which can be changed at a defined interval by deriving a new key to avoid a correlation of service usage. Furthermore, it provides an additional mechanism for access control. If the server tears down the previous custom onion service of a client each time it derivates a new temporary key and provisions new custom onion service, an adversary that gets into possession of that temporary key can only monitor the service that is bound to it. The adversary can not obtain the temporary key of a new time interval, neither can he obtain the temporary key of a previous time interval due to future secrecy.

Nevertheless, this approach does have a number of implications. As mentioned in the description of phase one, both the client and the server must agree on domain parameters in advance. This goal can be achieved by specifying the parameters in the protocol definition since they must not be kept secret. A more serious issue lies in the synchronization. Both participants must synchronize the derivation of new temporary keys, otherwise the client might use an address for a service that is no longer available or the server might offer a service that is no longer used. This synchronization is partly accomplished by specifying a time interval for the derivation of new temporary keys in the protocol definition. The interval by itself does not solve the issue. Fixed time points that define the start of an interval are required, which have to be identical for both systems. Furthermore, additional synchronization messages are needed in case one participant is offline during the beginning of a new interval.

- **Random Assignment:**

  An alternative approach is the randomized assignment of custom onion addresses. This is based on the premise that when a new onion service is provisioned, a new master keypair is created for each service, which cannot be associated with the keypair of other services. In the context of the UOSP this means that each time a new data address is requested, a dedicated onion service is provisioned, whose onion address is only dependent on the master identity public key and the version of the onion service and cannot be linked with previously created data addresses or to the introduction address of a service. This onion address is used as data address and can be considered random, as there are, beside the version, no parameters that influence its generation.

  This requires neither key exchange nor key derivation or synchronization, which reduces the complexity of the protocol since the assignment, which can be reduced to an onion service provisioning, is entirely handled by Tor.

Both options offer similar properties in terms of unlinkability. In both approaches, data addresses cannot be linked to other data addresses or introduction addresses. The *Key derivation* approach has the advantage that new data addresses can be generated without re-contacting the introduction address, but it has a significantly higher inherent complexity. As the requirements for unlinkability are met by both approaches, the *Random Assignment* is used due to its simplicity and lightweights.

4.3.4 Protocol Definition

The individual components of the protocol, as well as the basic procedure for the creation of unlinkable onion services have been described in the previous sections. In this section, a formal definition of the protocol will be presented. According to Holzmann et al. [44] a protocol specification consists of the five following components:

- "The service to be provided by the protocol"
- "The assumptions about the environment in which the protocol is executed"
- "The vocabulary of messages used to implement the protocol"
- "The encoding (format) of each message in the vocabulary"
- "The procedure rules guarding the consistency of message exchanges"

In the course of this section these components are defined for the UOSP.

- **Service Specification**

  The purpose of this protocol is the provisioning of one or multiple per-client ephemeral onion services, whose addresses are referred to as data addresses. The protocol is intended for the use in a client-server architecture. A proxy arranged between client and server abstracts the complexity of the protocol from the user. The initial connection from client to proxy takes place unidirectionally using a fixed onion address referred to as introduction address, whose sole purpose is the initial communication between proxy and server. No payload data must be exchanged over this connection. The proxy maintains a cache to store the results of previous requests to the introduction addresses of services. This is mandatory in order to ensure unlinkability for future requests. Each entry in the cache consists of the issued data address, its validity and the introduction address that was used to issue this data address. Once the proxy receives a request from the client, it first checks whether there is a valid data address in the cache for the introduction address in the request. If this is the case, the valid data address is structured in a *UOSP Client Response* and is sent back to the client.

  If the proxy does not hold a cached address, it extracts the required data from the request and generates a *UOSP Request*, which it then forwards to the server. The server provisions a new ephemeral onion service and responds to the proxy with the public key of the newly provisioned onion service encoded as its onion address. Additionally, the validity of the generated address is transferred as metadata. This validity set by the server controls how long a service can be accessed via its address. After the validity period has expired, the server tears down the onion service, which is thus no longer accessible. The proxy confirms the reception of the public key and its metadata by sending an acknowledgement message in form of a *UOSP Ack* to the server. Attached to this acknowledgement is a hash of the previously received public key. The server in turn hashes the public key of the previously provisioned onion service and is now able to compare both values.

If these values do not match, the integrity of the public key sent cannot be verified and it can be concluded that an error has occurred either in the transmission path from server to proxy or from proxy to server. To alert the proxy about this error, the server sends a *UOSP Error* to the proxy and tears down the previously provisioned service. The proxy relays this error to the client. If the two hash values match, the integrity of the public key is assured. To end the protocol cycle, the server sends a finalization note in the form of a *UOSP Fin* message to the proxy. The proxy caches the data address for future requests and sends a *UOSP Client Response* to the client. This response can either contain textual information about the newly provisioned data address or a protocol-specific redirect.

- **Environmental Assumptions**

  The environment in which the protocol is executed consists of a client, a proxy, a server and a transmission channel. It is assumed that the proxy is local to the client, which implies that the client's onion proxy and the proxy are located on the same system. Nevertheless, the operation of a remote proxy would be possible, but would not serve its purpose in terms of unlinkability, since information on the client's service usage could no longer be managed locally. As this protocol operates on the application layer in the Open Systems Interconnection model (OSI model), segmentation, flow control, congestion control and transport of protocol messages is performed by the underlying layers. This underlying layers assure that packets arrive at their destination in the correct order without data corruption. It is assumed that all components are online during a protocol cycle. Any archiving of protocol messages for later transmission is not considered. Both the proxy and the server must have a mutual consent on the time zone, which is used for temporal functions that are dependent on time.

- **Protocol Vocabulary**

  The protocol vocabulary specifies eight distinct types of messages:

  - **UOSP Request:**

    Requests the provisioning of a new temporary data address. The required introduction address is extracted from the UOSP client request.

  - **UOSP Response:**

    Response to a UOSP request. The payload of this response contains the encoded public key of the newly created data service, as well as metadata.

  - **UOSP Client Request:**

    Request from the client to the proxy. This requests must contain a valid destination address. The destination address can be an arbitrary address including introduction addresses and data addresses.

  - **UOSP Client Response:**

    Response to a UOSP client request that includes the data address of the newly provisioned data service. This response can either take a textual form or be implemented as a protocol-specific redirection.

- **UOSP Ack:**

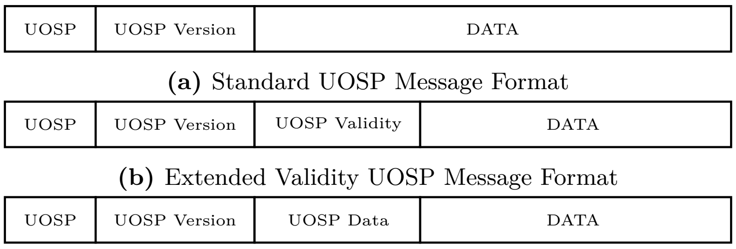
  Acknowledgement of the reception of a UOSP response, which includes the hash of the encoded public key received with the UOSP response.
- **UOSP Fin:**

  Indicates the successful completion of a protocol cycle.
- **UOSP Error:**

  Signals a corruption of the integrity of the transmitted encoded public key on the transmission path between proxy and server.

- **Message Format**

  Every UOSP message contains at least the UOSP specifier, which indicates whether the message is UOSP related or not and the UOSP version, which currently is 0.1. This basic message structure is referred to as standard message format. Dependent on the protocol message sent, the extended validity UOSP message format or the extended data UOSP message format is used. The extended validity format includes the validity of a data address transferred through the message, whereas the extended data format contains an indicator of whether data is transferred over a UOSP connection or not. The menagerie of message formats is depicted in Figure 4.3. The optional payload in all three message formats is stored in the data field.



(a) Standard UOSP Message Format

(b) Extended Validity UOSP Message Format

(c) Extended Data UOSP Message Format

Figure 4.3: UOSP Message Formats

The standard message format is used by the UOSP Request, UOSP Ack, UOSP Fin and UOSP Error. The extended validity format is used in UOSP responses. The UOSP client request and UOSP client response are not specified, as the client request is dependent on the protocol used (e.g. HTTP) and the client response should be adjustable to the users' environment. The extended data UOSP format is soleley used for data transfer using data addresses.

- **Procedure Rules**

  1. The resource a client wants to access must be provided by an onion service and thus have an onion address.
  2. The proxy must store previous requests to an introduction address and the responses containing the data addresses for these requests.

3. If the client requests an introduction address for which a data address has been provisioned previously, the cached address must be used. If the proxy does not hold a cached address, it must send a request to the server.
4. The cache of the proxy must delete entries that are invalid. This validity check must be carried out either periodically or with each request.
5. If the server receives a valid request for an introduction address associated with the server, it must provision a new onion service with its associated data address.
6. The response to a valid UOSP request must contain a valid data address for a temporary data service.
7. Every UOSP response must be acknowledged. This acknowledgement is used to ensure that the response has been received. To each acknowledgement the hash value of the data address must be appended in order to assure the integrity of the received address.
8. If the integrity check fails, the server must alarm the other participants by sending an error message.
9. The server must maintain all provisioned data services within their validity period.
10. If services exceed their validity periods, they must be teared down by the server. It is therefore mandatory that also the server maintains a cache storage for provisioned services and their validity.
11. The protocol ends by either a UOSP Fin or a UOSP Error message.

A complete overview of the UOSP can be obtained in the Unified Modeling Language (UML) sequence diagram in Figure 4.4.

4.4 Proof of Concept

To demonstrate the feasibility of the unlinkable onion service protocol proposed in section 4.3, a Proof of Concept (PoC) is implemented, which consists of a proxy module and a server module. In principle, the protocol is applicable for all TCP-based communication, however, for this implementation, the Hypertext Transfer Protocol (HTTP) was chosen, as this is apart from Skynet (botnet) the most common used protocol for onion services [45].
In the subsequent section, the technical components used for both the proxy and the server module are delineated. The implementations of the server and proxy module are described in detail in separate sections.

4.4.1 Technical Building Blocks

4.4.1.1 Tor Stem

Both the proxy and the server module are implemented with *Python* 3.7, due to the availability of the Tor *Stem* library [46]. Stem is a controller library written in Python, that enables communication with a Tor binary using the Tor control protocol. The library is composed of three different API modules at its core [46]:

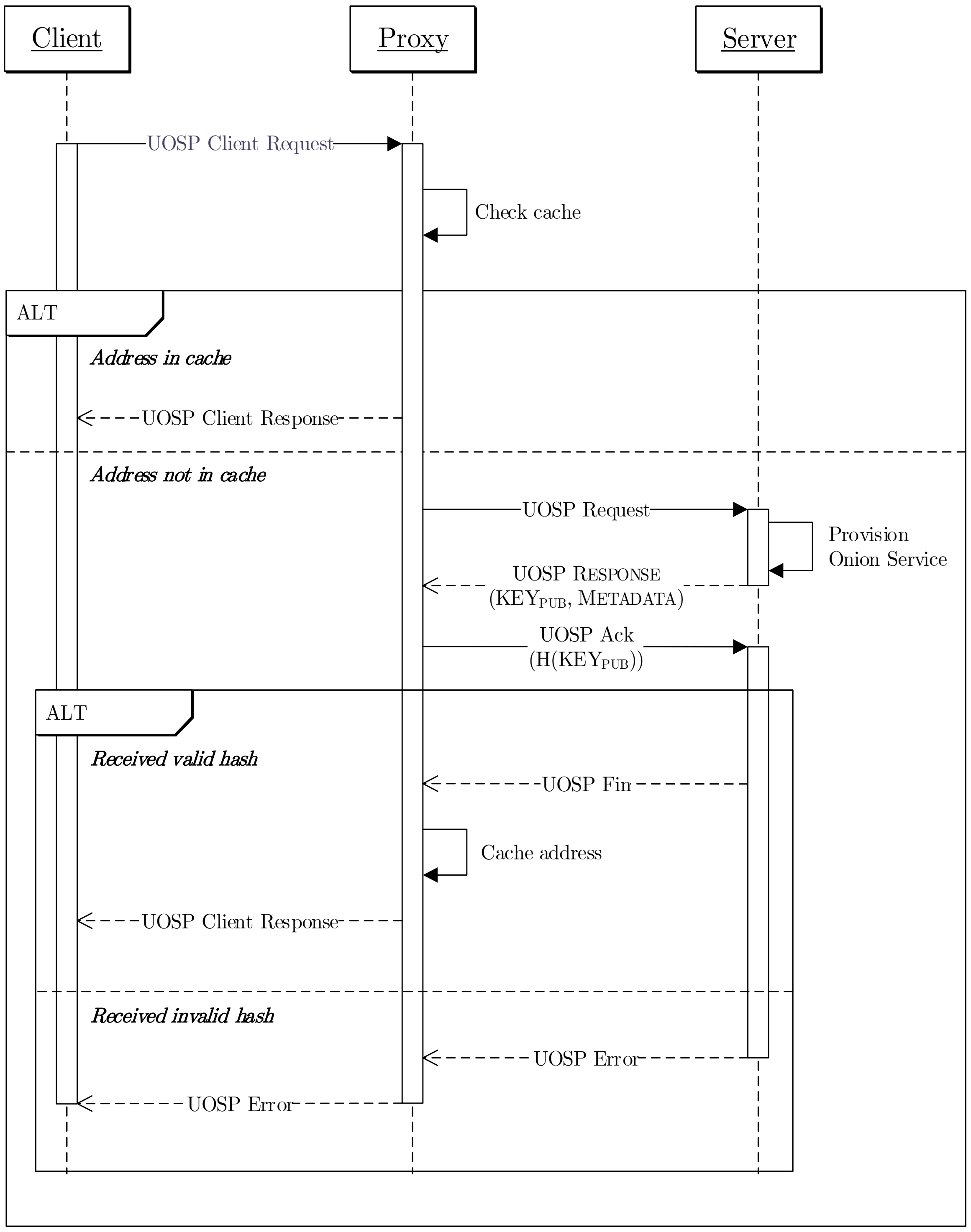


Figure 4.4: Unlinkable Onion Service Protocol Sequence

- **Controller:** The controller module enables authenticated communication with the Tor control socket and abstracts low-level connection details.
- **Descriptor:** The descriptor module can be used to retrieve, manage and generate a variety of data structures used in Tor. These include server descriptors, micro descriptors, network status documents, directories information and many more.
- **Utilities:** Helper functions of the two modules controller and descriptor are bundled in the utilities module. These include functions for retrieving system information, handlers for the textual configuration files, helper functions related to IP communication, string tools and functions for working with the underlying system.

In the course of this thesis, only the Controller module and its helper functions from the utilities module are used.

4.4.1.2 Socket Server

To implement the communication between proxy and server, the Python module *socketserver* is used. This module is a framework for creating network servers, which provides classes for the synchronous processing of network requests for different protocols and connection types. The processing of a request is thereby performed by two different classes. The *server* classes take care of the low-level communication tasks like listening on a socket or accepting connections. The protocol-specific processing, is performed by the *handler* classes. Their tasks include, for example, processing and interpreting incoming data and sending responses back to the requester. The advantage of dividing the responsibility for processing a request between communication-specific and protocol-specific classes is their re-usability. This re-usability enables users to utilize existing server classes for basic communication (e.g. TCP) without major changes and to adapt the protocol-specific handler to the respective application. [47]

A socket server is constructed by instantiating an object of a server class. Thereby, the address to be listened to by the server and a *request handler* class, which is called when a request is processed, must be passed as parameters. All request handler classes are based on the *BaseRequestHandler* super-class, which defines three interfaces, which must be implemented by handler classes inheriting from it: [47, 48]

- **Setup:** The setup method is called prior to the handle method. It prepares the request handler for the request by performing all initial actions required.
- **Handle:** The handle method carries out all the real work required to handle a request. This contains parsing the request, processing it and sending a response.
- **Finish:** The finish method is called after the handle method completed and performs a clean-up of anything created during the setup.

For both stream-based and datagram-based traffic, there are pre-defined base classes that inherit from the *BaseRequestHandler* and have already implemented the setup and finish methods so that only the actual request handling must be supplemented. As mentioned before, requests are processed synchronously. This means that a request must be completed before the next request can be processed, which implies that parallel processing of requests is not possible. To overcome this limitation, mix-in classes can be used to enable asynchronous processing in the form of a separate thread per request (*ThreadingMixIn*), or a separate process per request (*ForkingMixIn*). [47, 48]

4.4.1.3 Caching Database

As stated in the protocol definition in section 4.3.4, the data address of a newly provisioned temporary data service must be stored on both the server and the proxy. To avoid complications through inter thread communication or inter process communication, a central database is used to store the addresses. This database should generate as little overhead as possible. Therefore, renowned database management systems such as *MySQL* or *PostgreSQL* are excluded, as only a fraction of the functions offered are required.

The alternative lightweight database approach used for this prototype is the Zope Object Database (ZODB) [49]. As the name suggests, this is an object database management system, which was originally developed for the *Zope* web application server. ZODB is entirely written in Python and can therefore be seamlessly integrated in the code, as no separate language for database operations is used. Furthermore, no object-rational mapping is necessary, since the database can directly persist Python objects. This persistence does not have to be manually triggered by the user, nor any code is needed to read or write objects. The persistent objects are attached to a container, which works similarly to a Python dictionary. This dictionary is referred to as the *root* of the database. Every object that can be pickled into a standard serial format can be stored in the database. [49]

4.4.1.4 State Machine

Both the proxy and server module of the PoC are implemented using a finite state machine. This enforces the compliance of the implementation to the protocol specification, since only defined transitions are possible. In this prototype the light-weight, object-oriented Python implementation *transitions* is used [50].

4.4.2 Proxy Module Implementation

The basic operation of the proxy module entails that the client sends an HTTP request to the proxy, which evaluates the request. If the requested address is a UOSP introduction address, the proxy checks whether it has already stored an associated UOSP data address from previous requests. In this case, it sends this address as a response to the client. Otherwise, the proxy forwards the request to the server. The server provisions a new UOSP data service and sends its address to the proxy. The proxy forwards this address to the client and caches it for future requests. A detailed description of the protocol sequence can be found in section 4.3.4. The implementation of the proxy module is described on the basis of its finite state machine in Figure 4.5. For the sake of readability, generic names were used for states and transitions, which are described in Table 4.1.

$$\begin{tikzcd}
\text{start} \arrow[r] & S_0 \arrow[r, "a_0"] \arrow[d, "a_4"] & S_1 \arrow[r, "a_1"] \arrow[d, "a_5"] & S_2 \arrow[r, "a_2"] \arrow[drr, "a_6"] & S_3 \arrow[r, "a_3"] \arrow[dr, "a_7"] & S_4 \\
 & S_5 & S_6 & & & ERR
\end{tikzcd}$$

Figure 4.5: Finite State Machine of the UOSP Proxy Implementation

S_0	*S_1*	*S_2*	*S_3*	*S_4*	*S_5*	*S_6*	*ERR*
Initial state	Client request sent	Server public key received	Client ack sent	Server Fin received	Cache address used	Data response received	Error

a_0	*a_1*	*a_2*	*a_3*	*a_4*	*a_5*	*a_6*	a_7
Send request	Receive public key	Send UOSP Ack	Receive server Fin	Use cache address	Receive data response	Public key malformed	Server hash invalid

Table 4.1: Proxy Finite State Machine Legend

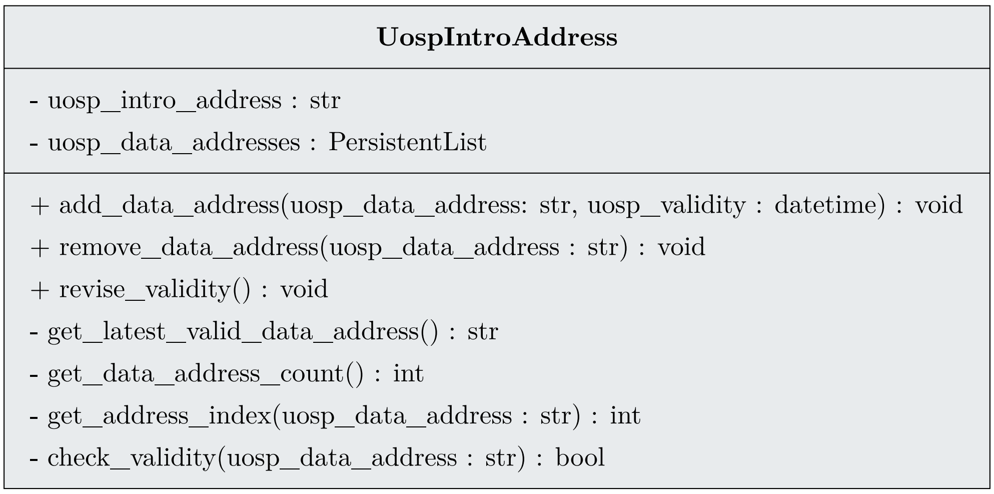
In the subsequent chapters, the implementation of the individual states and their transitions are described in detail. The complete source code of the proxy module can be found on GitLab [51] and in Annex A.

4.4.2.1 Initial State

To be able to receive requests from the client, the proxy uses the TCP server class of the socktserver module described in section 4.4. As this class processes request synchronously, the optional mix-in class ForkingMixIn is used. As a result, a new process is forked for each request, which can be handled asynchronously. The ThreadMixIn class, which would create a new thread for each request, was not used because each individual request needs to be isolated to avoid protocol failures. During the forking of a process, a state machine is automatically created for each request handler. This state machine is used throughout the protocol execution to keep track of the state of the process. Starting from the initial state, there are two possible transitions.

In order to decide which of the two possible transitions shall be carried out, the HTTP request received from the client must be analysed. For this purpose, the Python module *http-parser* [52] is utilized, which automatically parses the request and maps its data structure to an object. From the request header, the original destination address and its port are required. In this proof of concept implementation, it is assumed that the proxy only handles client requests whose top-level domain of the destination address is *.onion* and whose destination is port 80. HTTP requests whose pattern differ from this are forwarded to their intended destination, but are not considered part of the protocol.

In the next step, the proxy examines whether a UOSP data address has previously been issued for the UOSP introduction address used in the request and whether this data address is still valid. For this purpose, the proxy maintains an object database as described in section 4.4. This database is referred to as *cache* in the future. The class diagram of the objects stored in this cache is depicted in Figure 4.6:



```
UospIntroAddress

- uosp_intro_address : str
- uosp_data_addresses : PersistentList

+ add_data_address(uosp_data_address: str, uosp_validity : datetime) : void
+ remove_data_address(uosp_data_address : str) : void
+ revise_validity() : void
- get_latest_valid_data_address() : str
- get_data_address_count() : int
- get_address_index(uosp_data_address : str) : int
- check_validity(uosp_data_address : str) : bool
```

Figure 4.6: Proxy Database Objects

For each UOSP introduction address not known to the proxy, an instance of the class *UospIntroAddress* is created. The properties of these instances are the introduction address and a list of associated data addresses. The data type used for *uosp_data_addresses* is specific to the implementation of the ZODB object database used. If the Python list datatype were used for this property, changes to the list would not be recognized by the ZODB and thus would not be persisted. The UospIntroAddress class provides methods for adding, removing and revising the validity of UOSP data addresses. To check whether valid UOSP data addresses are stored on the proxy for a UOSP introduction address, first a revision of the existing addresses is carried out using *revise_validity*. Subsequently, the number of stored UOSP data addresses is checked using *get_data_address_count*. If this number is greater than zero, the cache of the proxy contains a valid UOSP data address. In this case, transition a_4 is triggered, otherwise the proxy has not stored a valid UOSP data address and must make a request to the server. For this purpose, a socket connection is established with the server. Although for this prototype implementation HTTP is used on the proxy side, the traffic between proxy and server module is encrypted using the python *ssl* module to avoid the leakage of data addresses. As the server functionality is provided as an onion service, the request must be done within the Tor network. In this proxy implementation Tor version 0.4.3.5 is used, whose configuration can be found in Appendix A.8.

By default, Tor automatically starts a SOCKS proxy on port 9050 if it is not manually deactivated [53]. This SOCKS proxy is used by the proxy module to be able to communicate over the Tor network. The proxy module sends a request to the server using the standard UOSP message format illustrated in Figure 4.3a, which triggers transition a_0.

4.4.2.2 Cache Address Used

The program execution transitions into this state if the proxy has a valid UOSP data address for a UOSP introduction address in its cache. This UOSP data address is sent to the client using the extended validity UOSP message format illustrated in Figure 4.3b. The payload of this message is an HTML page advising the user to use the UOSP data address provided. Alternatively, a HTTP redirect to the UOSP data address can be sent, which is carried out automatically by a browser, for example. As this is a final state, the protocol ends after submitting the response to the client and the process is stopped.

4.4.2.3 Client Request Sent

After sending a UOSP request the proxy enters an exchange loop where it waits for the response of the server. If a response is received, it is parsed in the first step using the Python module *http_parser*. This separates the HTTP headers from the payload and examines which type of response was retrieved, which results in three options:

1. The response has the form of an extended data UOSP message illustrated in 4.3c and thus is the response to a request that used a UOSP data address. The exchange of user data, although included in this state machine, is not part of the UOSP.
2. The response has the form of an extended validity UOSP message illustrated in 4.3b and thus is the response to a request that utilized a UOSP introduction address.
3. The response does not have a UOSP header and thus is a response to an onion service not capable of performing UOSP. The data received is relayed to the client, although the client data exchange is not part of the UOSP and is therefore not shown in the state machine.

In the first case, transition a_4 is triggered, whereby the state *Data response received* is reached. The reception of a public key as in the second case triggers transition a_1 and leads to the state *Server public key received.*

4.4.2.4 Data Response Received

In case of a response to a request that used a UOSP data address, the payload of the response must be forwarded. The payload is extracted from the response and sent to the client in the form of a standard UOSP message, whose structure is shown in Figure 4.3a. As this is a final state, the protocol ends after submitting the response to the client and the process is stopped.

4.4.2.5 Server Public Key Received

Once the public key has been received, its validity is checked in the first step. This can be implemented very simply, as the public key is available in its encoded form as onion service V3 address, which must contain exactly 56 characters. According to the rendezvous specification [25], it should additionally be checked whether the key contains a torsion component, as this would allow attackers to create multiple onion addresses with the same key for a single service. As this is only a prototype implementation, this check is not implemented and can be considered as future work. If the public key is malformed, transition a_6 is triggered which results in the error state. Otherwise, it is verified in the next step that the integrity of the transferred key was not violated during the transmission. For this purpose, the obtained key is hashed with the *SHA-256* hash function, which is provided by the Python module *hashlib* [54]. The resulting hash value is appended as payload to a standard UOSP message, whose structure is illustrated in Figure 4.3a. This message is referred to as UOSP Ack in the protocol and triggers transition a_2.

4.4.2.6 Client Ack Sent

After the UOSP Ack has been sent to the server, the proxy waits for its response in the form of an extended Data UOSP message. If the integrity validation fails, meaning that the hash value the server created doesn't match the hash value provided by the proxy, the response contains the payload *ERR*, which triggers transition a_7 and leads to the error state. Otherwise, the payload is *FIN*, which triggers transition a_3 and results in the state *Server Fin received.*

4.4.2.7 Server Fin Received

If the proxy receives the protocol message *Fin*, a new UOSP data address has been successfully generated. This address is sent to the client in the form of an extended validity UOSP message. As mentioned in the description of the state *Cache address used*, an HTTP redirect can be sent alternatively. Finally, the UOSP data address is cached. The datastructure used for the storage of the data addresses is depicted in diagram 4.6. If no *UOSPIntroAddress* object with the UOSP introduction address exists yet, a new object is instantiated. Subsequently, the UOSP data address is added using the *add_data_address* method. As this is a final state, the protocol ends after submitting the response to the client and the process is stopped.

4.4.3 Server Module Implementation

The primary function of the server module in the UOSP consists in the provisioning of temporary data services. These provisions are triggered by the proxy module through a specially formatted HTTP request to a UOSP introduction address. After validating the responsibility for the received request, the server module connects to the Tor binary, which is used to provision an ephemeral onion service. The onion address of this newly created onion service is stored in the server cache to be able to manage all the services provisioned. Subsequently, the address is transmitted to the proxy module using a UOSP message. The proxy module receives the address and responds with its hash value. The server module compares the received hash value with the hash value of the onion address generated by itself and determines whether the integrity has been compromised on the transmission path. If this is the case, it sends an error message to the proxy module. Otherwise, the server responds with a message that indicates the end of the protocol cycle. The implementation of the server module is described on the basis of its finite state machine in Figure 4.7. For the sake of readability, generic names were used for states and transitions, which are described in Table 4.2.

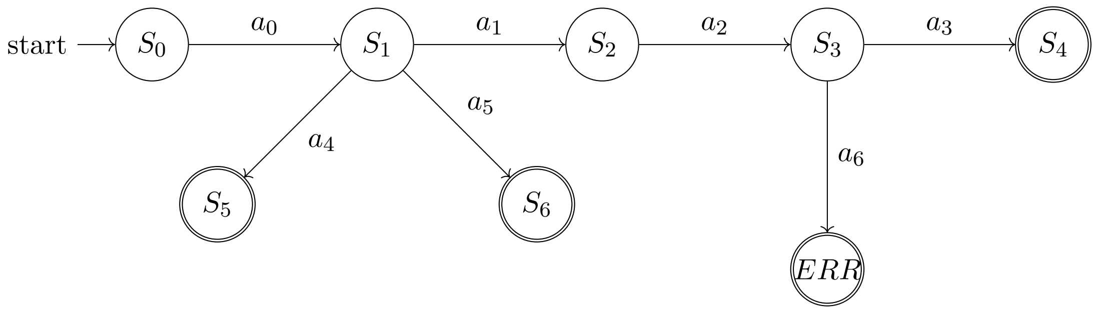

$$\begin{tikzcd}
\text{start} \arrow[r] & S_0 \arrow[r, "a_0"] & S_1 \arrow[r, "a_1"] \arrow[dl, "a_4"] \arrow[dr, "a_5"] & S_2 \arrow[r, "a_2"] & S_3 \arrow[r, "a_3"] \arrow[d, "a_6"] & S_4 \\
& S_5 & & S_6 & ERR &
\end{tikzcd}$$

Figure 4.7: Finite State Machine of the UOSP Server Implementation

*S*_0	*S*_1	*S*_2	*S*_3	*S*_4	*S*_5	*S*_6	*ERR*
Initial state	Client request received	UOSP response sent	UOSP Ack received	Server Fin sent	Data response sent	Legacy HTTP response sent	Error

*a*_0	*a*_1	*a*_2	*a*_3	*a*_4	*a*_5	*a*_6
Receive request	Send UOSP response	Receive UOSP Ack	Send server Fin	Send data response	Send legacy data response	Public key malformed

Table 4.2: Server Finite State Machine Legend

In the following, the implementation of the individual states and their transitions are described in detail. The complete source code of the server module can be found on GitLab [51] and in Annex A.

4.4.3.1 Initial State

In order to be able to receive and respond to HTTP requests from the proxy, the server module must maintain an HTTP server, which needs to be provided as onion service. For the provisioning of the HTTP server, the Python module *http.server* [54] is used, which is a sub-class of the *socketserver* base classes described in section 4.4. This module includes a special sub-class of the *BaseRequestHandler* that allows to handle GET and POST requests in a straightforward way. As with the proxy module, the mix-in class *ForkingMixIn* is used, which creates a new process for each request. In addition, the python module *ssl* is used to encrypt the traffic between proxy and server module. It must be noted, that the ssl configuration used is intended for experimental purposes only and should not be used in a production environment. Since the server must be reachable via the tor network, the server module needs to launch a Tor binary. In this server implementation, Tor version 0.4.3.5 is used whose configuration excerpt can be found in Listing 1. The complete configuration is provided in Annex A.5.

```
ControlPort 9051
HashedControlPassword 16:0717EF71CDF228DE60E93242A4AFE681FB525080DC1A330A103FAD9782
```

Listing 1: Excerpt from the Tor Configuration of the Server Module

In contrast to the Tor configuration of the proxy, the configuration of the server activates the control port and protects it with a hashed password. To make the HTTP server accessible via an onion service, the Tor Stem library is utilized. For the usage of Stem, auxiliary functions were developed, whose source code can be found on GitLab [51] and in Annex A.2. The headers of these auxiliary methods are provided in Listing 2.

```python
def establish_connection(address=CONTROL_ADDRESS, port=CONTROL_PORT, password=None)
def create_ephemeral_hidden_service(controller, source_port, target_port, version)
def create_hidden_service(controller, path, port)
```

Listing 2: Header of the Tor Stem Auxiliary Methods

The method *establish_connection* is used to connect to the Tor control socket, while the methods *create_ephemeral_hidden_service* and *create_hidden_service* are used to create onion services. Ephemeral onion services are bound to the control connection through which they were created, which means that the ephemeral onion services in this implementation would be teared down after the protocol cycle is finished. To overcome this issue, the Tor Stem flag *detached=True* is used, which detaches the service from the controller so that it can continue to exist after the end of the control connection.

After the provisioning of its introduction service, the server module resides in its initial state and waits for a client GET request to arrive. A list of UOSP introduction addresses provided by the server module is maintained in order to be able to distinguish them from UOSP data addresses later on. The reception of a request triggers the transition a_0, which leads to the state *Client request received*.

4.4.3.2 Client Request Received

Once the UOSP request has been received, it must be ensured that the connection is not terminated after the request has been processed and the response has been sent, since the server module must wait for the UOSP Ack. In order to avoid the connection tear-down, the argument *close_connection* of the handler class must be set to false. In the next step, the server examines whether the request received is a UOSP message, which is characterized by the presence of specific UOSP headers, as illustrated in Figure 4.3. If this is not the case, the message is a legacy HTTP request and the server sends an according response in the form of an extended data UOSP message to the proxy, which triggers transition a_5. In this prototype, the data addresses can only be accessed via the proxy, since their direct utilization could lead to a dis-balanced usage of certain data addresses in terms of frequency and could potentially render the protocol useless.
If, on the other hand, the request contains the required UOSP headers, a further distinction must be made:

1. The requested address is one of the UOSP data addresses of the server module.
2. The requested address is one of the UOSP introduction addresses of the server module.

In the first case, the server must respond with the web page the client intended to reach. Therefore, an extended data UOSP message with the service's user data is created and sent to the proxy. This triggers transition $a4$, which results in state *Data response sent*. It must be pointed out that the exchange of data (both legacy and data requests) is not part of the UOSP, but has been implemented in this prototype.

In the second case, a new UOSP data service must be provisioned. For this purpose, a connection with the Tor control socket is established and an ephemeral onion service is provisioned using the auxiliary method excerpted in Listing 2. Each provisioned address is automatically assigned a validity by the server, which is currently set to 24 hours. A detailed analysis of the impact of the length of the validity period on the unlinkability of onion service is considered future work. After the validity period has expired, the service is torn down, which requires the server module to store information on all provisioned UOSP data services. For this purpose, the server maintains an object database as described in section 4.4.1. This database will be referred to as *cache* in the future. The class diagram of the object stored in this cache is depicted in Figure 4.8.

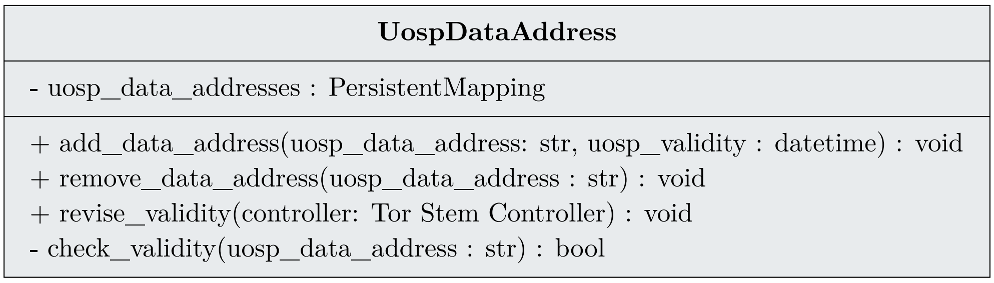

UospDataAddress
- uosp_data_addresses : PersistentMapping
+ add_data_address(uosp_data_address: str, uosp_validity : datetime) : void
+ remove_data_address(uosp_data_address : str) : void
+ revise_validity(controller: Tor Stem Controller) : void
- check_validity(uosp_data_address : str) : bool

Figure 4.8: Server Database Objects

In contrast to the proxy, the server module does not create a new object for each UOSP introduction address, as the distinction is not relevant when invalidating UOSP data services. For this reason, there is only one instance of the *UospDataAddress* class that is used by all processes. The only property of this object is a dictionary like data type that contains the provisioned UOSP data addresses as keys and their validity as values.

In a separate thread, the server periodically checks whether the validity period of one of the data services in the *uosp_data_addresses* list has exceeded by utilizing the method *revice_validity*. A handle for communication with the Tor control socket is passed to the method so that exceeded services can be teared down immediately. Finally, the service is deleted from the list using *remove_data_address*.

In case of a newly provisioned data service, its onion address is added to the database using the method *add_data_address*. Subsequently, the server prepares an extended validity UOSP message, whose structure is illustrated in Figure 4.3b and appends this address as payload. Finally, this message is sent to the proxy, which triggers transition $a1$.

4.4.3.3 Data Response Sent & Legacy HTTP Request Sent

After sending the data response to the proxy, the work for the server module is done. As this is a final state, the protocol ends and the process for handling the request is stopped.

4.4.3.4 UOSP Response Sent

Once the UOSP response has been sent to the proxy, the server module waits for its acknowledgement. Since data is transmitted with this response, the server expects the usage of the HTTP POST method rather than the HTTP GET method. The reception of that acknowledgement triggers transition a_2, which results in the state *UOSP Ack received.*

4.4.3.5 UOSP Ack Received

In the first step, the received acknowledgement is divided into header and body. Since the *http.server* Python module was used for providing the HTTP server, no additional tools or modules are necessary in order to achieve this separation. It is assumed that the body of the acknowledgement solely contains the hashed public key that has been sent to it in the previous state. To assure the integrity of the transmission channel, the server also calculates the hash value of the address of the provisioned UOSP data service. Similarly to the proxy, a *SHA-256* hash is generated using the Python module *hashlib*. The hash value contained in the acknowledgement is then compared with the generated hash value. If these values differ, the encoded public key of the UOSP data service has been changed during transmission. This triggers the transition a_6, which leads to the error state. Otherwise, the server sends a response in the form of an extended data UOSP message to the proxy that indicates the successful completion of the protocol cycle. This trigger transition $a3$, which results in the final state *Server Fin sent.*

4.4.3.6 Server Fin Sent

After sending the finalize message to the proxy, the work for the server module is done. As this is a final state, the protocol ends and the process for handling the request is stopped.

5 Performance Analysis

As stated in the related work section, Köpsell et al. [5] discovered that the increase in latency of an anonymized service is linearly related to the drop-out rate of users. Based on these findings, Wendolsky et al. [6] were able to define the tolerance level of inexperienced users with regard to the latency of a service at about four seconds. This implies that an initial response to a user request must not exceed this duration, otherwise the usability of the service is limited. That also applies to the *Unlinkable Onion Services Protocol* (UOSP) specified in section 4. Additional latency is added to the communication through the provisioning of UOSP data services and the exchange of their addresses.

This section examines whether the increase in the latency of an onion service through the use of the UOSP can be reconciled with the latency limit defined by Wendolsky et al. [6]. Since the provisioning of UOSP data services will most likely account for the majority of the additional latency, it is necessary to find a method to measure the duration of the provisioning of an onion service. Several temporal measurements of onion services have already been carried out [10, 12, 14], but these measurements focused primarily on the access-time of services and were designed for the out-of-date V2 onion services. This thesis fills this gap by designing and implementing a system to measure the performance of onion services with respect to their provisioning time. In the following sub-sections, the methodological approach, the architecture and the implementation of this system are described in detail. To conclude, the results of the implementation and its impact on the usability of the UOSP will be discussed.

5.1 Methodological Approach

In order to be able to measure the provisioning time of an onion service it must be defined from which point on a service is considered to be provisioned. For this purpose, it is necessary to look at the individual steps that are involved in the provisioning. Initially, the service randomly selects Tor nodes that shall act as its introduction points. It builds anonymous three-hop circuits to those nodes and utilizes the *Extensible ESTABLISH_INTRO* protocol [25] to register them as introduction points for the service. Once it receives the acknowledgements for this registrations in the form of *INTRO_ESTABLISHED* cells, the onion service builds its descriptors, which contain, among other things, information on the established introduction points. After the completion of descriptor creation, anonymous circuits to the *Hidden Service Directories* (HSDirs) responsible for the service for the respective time period and replica are established. Taking into account both replicas and the two time periods under consideration (current and previous), there are a total of 16 responsible directories, to which a descriptor upload is initiated. Since the uploads are conducted in parallel, the order in which the uploads are started need not correspond to the order in which they are finished. Therefore, the successful completion of a specific upload event is not sufficient to determine whether the provisioning of the service has been completed. In principle, there are three different approaches to identifying the finalization of an onion service provisioning:

- The provisioning of an onion service is completed if all of its descriptor uploads are finished. This would imply that provisioning fails if one of the 16 uploads is not successful.

- The provisioning of an onion service is completed if the service can be accessed by clients, which indicates that at least one upload has successfully finished. This approach is used by the Tor stem library to determine whether the service was published or not.
- The provisioning of an onion service is completed if at least 50 percent of all uploads finished successfully. The sequence of the uploads is thereby not relevant.

In this work, a blend of two of the approaches described above is used. All service provisions in which one of the uploads has failed are counted as failed provisions. This is necessary to preserve the comparability of individual measurements. On the other hand, for the calculation of the overall provisioning time of an onion service, only the time required to complete 50 percent of the uploads is taken into account.

Another methodological consideration concerns the granularity of the measurements. For the measurement of the provisioning time, it would be sufficient to measure the total time from the creation of the service until the completion of the required uploads. Nevertheless, this would prevent the detection of any performance bottlenecks, which could be mitigated by measures and would significantly complicate troubleshooting in case of performance issues. For this reason, all sub-steps of the provisioning of an onion service shall be observed. A list of all measurable steps is provided in Table 5.1.

#	*Source*	*Target*	*Event*	*Measurable*	*Measured*
1	Service	Service	Started onion service provisioning	Yes	Yes
2	Service	Introduction Points	Started the creation introduction circuits	Yes	Yes
3	Service	Introduction Points	Finished the creation of introduction circuits	Yes	Yes
4	Service	Introduction Points	Sended *ESTABLISH_INTRO* cells	Yes	Yes
5	Introduction Point	Introduction Point	Registered introduction points	Yes	No
6	Service	Introduction Points	Received *INTRO_ESTABLISHED* cells	Yes	Yes
7	Service	Service	Started the creation of service descriptors	Yes	Yes
8	Service	Service	Finished the creation of service descriptors	Yes	Yes
9	Service	HSDirs	Started the descriptor uploads	Yes	Yes
10	Service	Service	Started the creation of upload circuits	Yes	No
11	Service	HSDirs	Finished the creation of upload circuits	Yes	Yes
12	Service	HSDirs	Finished the descriptor uploads	Yes	Yes

Table 5.1: Measurable Events During the Provisioning of an Onion Service

All actions in this table are referred to as events. The first event that takes place in the course of an onion service provisioning is the service creation. Thereby, two different types of services can be created. Non-ephemeral onion services are generated by either changing the in-memory Tor configuration or the Tor configuration file (*torrc*). In the latter case, the Tor binary must be reloaded in order to recognize the modification. For each non-ephemeral onion service, a new folder is created in the file system of the service host, which contains, among other things, the key material of the service, keys for client authentication and the host name of the service. On a restart of the Tor binary, all onion services contained in the Tor configuration are automatically re-provisioned. In contrary, ephemeral onion services are not created via the configuration file, but via the control socket of the Tor binary. For this purpose, for example, the Tor *Stem* library can be used. Ephemeral services exist only as long as their control connection exists. If the process through which the service was created using the control protocol is terminated, the service is also torn down. To decouple the services from their process, the attribute *detached* must be set to *True* when the service is provisioned. However, this has no effect on the persistence of ephemeral services. Restarting the Tor binary will not re-provisioning ephemeral onion services. Since both types of services can be created using the Tor control protocol, the creation timestamp of an onion service corresponds to the time the required control command was issued.

The second measurable event is the start of the creation of the circuits from the service to the chosen introduction points. If available, Tor tries to use existing pre-built circuits for that purpose, which only need to be extended by one hop [14]. This technique is referred to as *cannibalization* and decreases the time for the service provisioning, since the usage of a cannibalized circuit does not introduce further delay [14]. The completion of introduction circuit creation is measured by means of the third event. Through the measurement of the start and end times of introduction circuit creations, their duration can be computed. The fourth measurable event after the creation of the introduction circuits is the sending of the *ESTABLISH_INTRO* cells, which are needed to register the chosen Tor nodes as introduction points of the service. Once the Tor nodes receives the requests from the service, they carry out cryptographical validations and sanity checks and register themselves as introduction points. The duration of this registration can only be measured on the introduction points themselves, which means that, for the measurement, an introduction point would have to be operated and the Tor source code would have to be modified such that this introduction point is selected by the service. As this would not provide any significant new insights, this specific duration is not measured in the course of this thesis. The introduction points acknowledge the registration by sending an *INTRO_ESTABLISHED* cell to the service. This is understood as the completion of the establishment of the introduction points and is measured as the sixth event. Since the start and end times of the introduction point establishments are measured, its duration can be computed.

Using the established introduction points, the service generates its service descriptors in the next step. The start and the end time of this generation are captured in the seventh and eighth event, which allows the duration of this process to be measured. In the last step, the created descriptors are uploaded to the responsible HSDirs. For this purpose, circuits to the responsible directories must be established. Similarly to the introduction circuits, cannibalization can be used to speed up this process. It may seem odd that the event that indicates the start of the descriptor upload chronologically precedes the event for the creation of the upload circuits, but this event triggers the whole upload process, without which the circuits would not be created. Since the time gap between the ninth and tenth event corresponds to a median of about *0.00144* seconds, the start of the upload circuit creation is not measured, instead, the start of the descriptor uploads also marks the start of the circuit creation. The completion of the upload circuit creation is measured in event eleven and simultaneously indicates the start of the descriptor uploads. The uploads are asynchronous, which means that the sequence in which the uploads start does not necessarily influence the sequence in which they end. Completion of uploads is measured in the twelfth event. As the start and end times of the descriptor uploads are measured, its duration can be calculated. Additionally, the HSDirs for the individual descriptor uploads are recorded.

Finally, it must be determined how the events specified in Table 5.1 can actually be measured. Essentially, there are two possible approaches:

1. Tor offers an extensive log system, which can be enabled by appropriate entries in the *torrc* configuration file. Individual severity levels or functionality domains can be selected, whose log messages are then redirected to a file, the standard output or syslog, for example [53]. Therefore, one possible approach is to configure custom logging, which saves the required events to the chosen medium. On completion of provisioning, the log events from the selected medium are read out and evaluated.

2. All required events can be obtained through the use of the Tor control protocol. Thereby, a user or a program can register for specific event classes and then receive all log events corresponding to this class.

In the course of this work, the events are obtained via the Tor control protocol using the Tor Stem library, as this provides a considerably better handling in Python than reading from files or reading the standard output. In addition, each event class can be handled in its own thread, which enables parallel processing.

In principle, the focus of the measurement is set on V3 onion services, as V2 onion services will no longer be supported in the foreseeable future [15]. Nevertheless, currently more than 170,000 V2 onion services are still running, which renders them a non-negligible group in terms of their high prevalence [41]. Therefore, V2 onion services are included in the measurements in order to be able to compare their temporal behaviour and performance with the current version. In addition, the influence of the method used to create the onion service (ephemeral and non-ephemeral) and the use of additional security features, as described in 3.2.5, is measured.

5.2 Architecture

In order to obtain measurement results that are ideally unaffected by possible sources of interference, these interferences must be identified in the first step. Since the components used to perform the measurements are most likely operated within an university or corporate network, they are subject to fluctuations in network performance dependent on the time of day and number of users. Furthermore, if the underlying hardware of the server on which the components are operated is shared with other applications, their utilization can influence the hardware's resource consumption and thus influence the measurement performance. Similarly, the Tor network, which is used for the provisioning of onion services, is subject to fluctuations in terms of bandwidth and circuit build times that could potentially have an impact on the measurement results. To prevent these short-term fluctuations in the networks and systems from having an effect on the measured values, numerous measurements must be performed at different points in time.

In order to achieve this, an architecture is required that automatically provisions onion services, measures the events that occur in this process and outputs them in a format that can be used for analysis. To avoid measurements influencing each other (e.g. through caching), each measurement must be carried out completely isolated from the other measurements. This requires an environment that can be reset or recreated for each measurement with no significant overhead. In order to achieve this, containers are used, which are inherently isolated from each other through operating system features. In addition to avoiding interfering influences, a significant advantage of using containers consists in the fact that faulty measurements do not influence other measurements in terms of a temporal delay.

The minimal set of components required for measuring the provisioning time of onion services using containers are depicted in Figure 5.1. In the following, the individual components and their mode of operation are described.

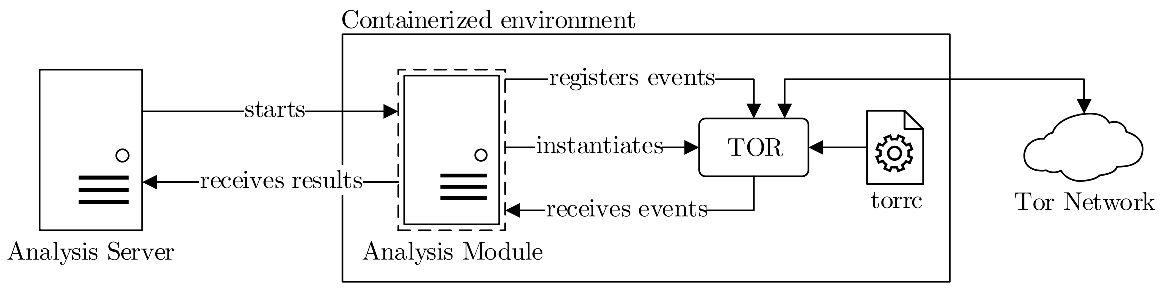


Figure 5.1: Performance Measurement Architecture

- **Analysis Server**

  The *Analysis server* constitutes the backbone of the measurement system. It periodically initiates a new measurement instance by starting a new measurement container. This container performs all necessary steps for the measurement and returns the result to the analysis server in a pre-defined format. Irrespective of the wording, the analysis server does not necessarily have to deployed on a separate physical server, but can be located on the same server as the host of the measurement containers.

- **Analysis Module**

  All steps necessary for measuring the provisioning time of an onion service are performed by the *Analysis Module.* In the first step, a new Tor process is created and Tor is bootstrapped using the configuration in the provided *torrc* file. Subsequently, the module connects to the Tor control socket using the Tor stem library and registers for several protocol events. When these events occur, the module receives notifications containing all information about the events. Once all event listeners are registered, a new onion service is created. Now the time values relevant for the measurement are received by means of the previously registered event listeners. As soon as the generation of the service and its measurement are completed, these time values are analysed and correlated. The result of this analysis is returned to the analysis server in a pre-defined format.

- **Tor Binary**

  The Tor binary represents the connection between the analysis module and the components of the onion service to be provisioned in the Tor network. To avoid distorting the measurements, it is important that no caching of any kind is performed on the side of the Tor binary. This is ensured by creating a new Tor process for each measurement.

- **Tor Network**

  Since onion services can only be provisioned and accessed in the Tor network, the analysis module and respectively the Tor binary must be connected to it.

The advantages of this architecture are its modularity and its ease of use. Since the analysis module is implemented as a container, it includes all components required for the measurement. A measurement process is initiated simply by starting the container, which returns the result in a format that can be read by a wide range of analysis tools. The most important configuration settings can be set with parameters that are specified when the container is started, without having to change the container. In addition, it is possible to replace the analysis server with another solution and thus integrate the analysis module into another software project.

5.3 Implementation

As described in section 5.2, the architecture for measuring the provisioning time of onion services consists of the three main components *Analysis Server*, *Analysis Module* and the *Tor* binary. In the following subsections, the implementation and usage of these components and all decisions made in their development process are described in detail. Since space is limited, only excerpts from the source code can be shown in the solution description. The complete source code of the implementation is provided on GitLab [55] and in Annex B.

5.3.1 Analysis Module

The analysis module forms the core of the measurement system. It is implemented in *Python* 3.7, due to the dependency on the Tor Stem library [46], which is used to measure the events listed in Table 5.1. A description of the structure of this library can be found in section 4.4.1. The analysis module consists of several Python files, whose structure is shown in Figure 5.2.

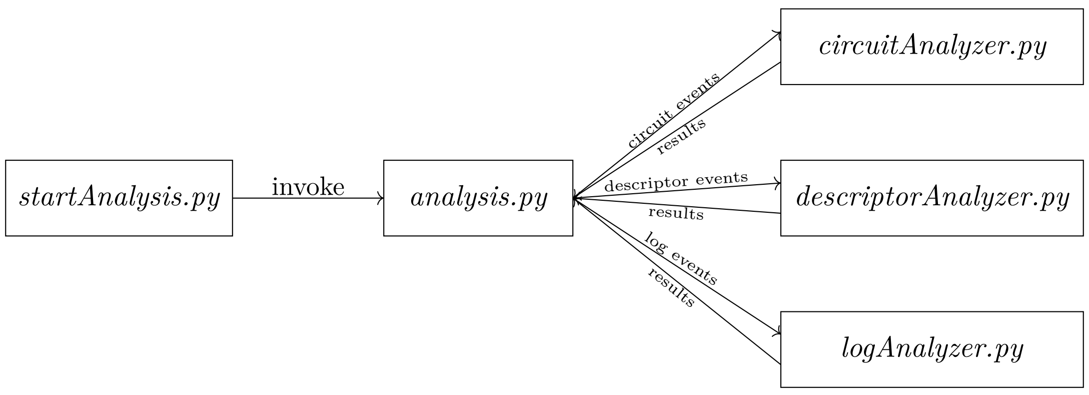


Figure 5.2: Analysis Module Program Structure

In order to be able to connect to the Tor control socket and ensure that there is no timing corruption, it is important to verify that the Tor process is fully bootstrapped. This verification is realized within the file *startAnalysis.py* by matching a predefined string sequence that indicates the completion of the bootstrap using regular expressions. The source code is provided in Annex B.5.

Once a match is found, the actual measurement activity is started by invoking the function *start* in the file *analysis.py*. The header of this function is shown in Listing 3.

```python
def start(version, ephemeral, timeout, num_intro_points)
```

Listing 3: Measurement Invocation

Thereby the main measurement settings are passed as function parameters. The parameter *version* specifies the version of the onion service to be provisioned and measured. Currently, the only recognized values are version two and version three. If *ephemeral* is set to true, the module provisions an ephemeral onion service. Otherwise, a non-ephemeral onion service is generated. The parameter *timeout* defines the maximum overall duration a measurement process may last before it is aborted. This is intended to prevent a failed or incomplete measurement from delaying the measurement procedure. The number of introduction points an onion service uses can be specified in the parameter *num_intro_points*. However, this value is only used to inform the module of how many introduction points are expected and has no influence on the number of the actual introduction points used, which can be set by manipulating the Tor source code.

In the first step of the measurement process, a connection with the Tor control socket is established using the *from_port* function of the Stem library, which returns a connection handle referred to as *controller*. Subsequently, this controller is used to register event listeners for certain groups of Tor events using the Stem function *add_event_listener*. As parameters, this function is provided with the event or group of events to be listened to and the name of the function to which the event or group of events should be relayed. The following groups of events are required to measure the onion service performance: [46]

- **CIRC**

  A *CIRC* event takes place when a circuit has changed. The content layout of the information contained in the event is shown in Figure 5.3. One of the most valuable fields in this structure with regard to the comprehensibility of the measured information is the *id*. This represents the unique identifier of a circuit, which can be used to track it. The *status* field provides information about the status of a circuit. The fingerprints and nicknames of all Tor relays currently used by the circuit are contained in the field *path*. The field *purpose* contains a description of the purpose of the circuit. Information on the other fields can be obtained from the Stem manual [46].

id	*status*	*path*	*build_flags*
purpose	*hs_state*	*rend_query*	*created*
reason	*remote_reason*	*socks_username*	*socks_password*

Figure 5.3: Structure of a CircuitEvent

All circuit events are forwarded to the function *circCallback* provided in the file *circuitAnalyzer.py*, whose source code is provided in Annex B.2. As described in Table 5.1, the start and end time of the introduction circuit creation and the end time of the upload circuit creation are measured. For this purpose, all circuits elements in relation to the onion service must be recorded. This is achieved by evaluating the fields of the circuit event. Due to the gradual nature of circuit building, it is necessary to extract the individual circuit relays from a circuit event and assign them to their corresponding circuit according to the circuit id in the *id* field. This is accomplished by extracting the circuit fingerprints from the *path* field of the event and assign the relay to the circuit it extended.

In order to be able to effectively access all the nodes of a circuit, a class for storing the individual circuit elements was developed, whose class diagram is provided in Figure 5.4.

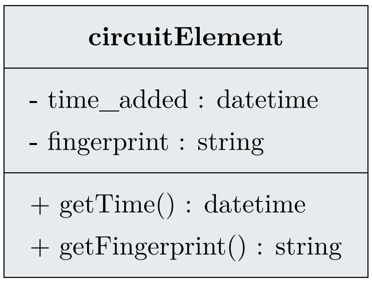


Figure 5.4: Circuit Element Class Diagram

Each node in the circuit is identified by its unique *fingerprint*. The time at which the node was added to the circuit is stored in the attribute *time_added*. This attribute is decisive for the subsequent time measurement. Both attributes can be obtained using a getter method.

To be able to assign the individual circuit elements to their circuits, a dictionary entry is created for each circuit, whose key is the unique circuit id. The values of these dictionary entries are lists to which the individual circuit elements are appended. In the end, all elements of a circuit can be retrieved by accessing the dictionary using the specific circuit id.

- **CIRC_MINOR**

  As the name suggests, the *CIRC_MINOR* events is triggered when minor changes to circuits are performed. The structure of the information contained in the event is illustrated in Figure 5.5. Since this is also a circuit event, the fields *id*, *path*, *build_flags*, *purpose*, *hs_state*, *rend_query* and *created* are identical to the *CIRC* event. The *event* field provides information about the minor change that has been carried out on the circuit. Thereby it is distinguished whether an existing circuit is cannibalized or the purpose of the circuit has changed. Information on the other fields can be obtained from the Stem manual [46].

id	*event*	*path*	*build_flags*
purpose	*hs_state*	*rend_query*	*created*
old_purpose		*old_hs_state*	

Figure 5.5: Structure of a CircuitMinorEvent

  This event is used to distinguish between a newly built or a cannibalized circuit by using only the fields *id* and *event*. For this, all circuit minor events are forwarded to the callback handler *circMinorCallback* provided in the file *circuitAnalyzer.py*. The procedure of the callback handler is identical to the procedure of *circCallback* described above.

- **HS_DESC**

  Any operations that affect the descriptors of an onion service trigger the *HSDescEvent* event, whose structure is depicted in Figure 5.6. The field *action* contains the description of the action carried out. An identifier of the Hidden Service Directory (HSDir) to which the descriptor is uploaded is contained in the field *directory_fingerprint*. Information on the other fields can be obtained from the Stem manual [46].

action	*address*	*authentication*	*directory*
directory_fingerprint	*directory_nickname*	*descriptor_id*	*reason*
replica		*index*	

Figure 5.6: Structure of a HSDescEvent

All descriptor events are relayed to the method *descCallback* provided in the file *descriptor-Analyzer.py*, whose source code is provided in Annex B.4. The events are necessary for the measurement of the duration of the descriptor creation and the descriptor upload. To distinguish the individual events, the field *action* is utilized. The action *CREATED* indicates that the onion service descriptor was just created, whereas the action *UPLOAD* and *UPLOADED* mark the beginning and the end of the descriptor upload. Since numerous descriptor events take place during the provisioning of an onion service, the events are stored separately and are correlated and analysed later on. For storing the events, an object structure similar to the structure of the circuit events is used, which is illustrated in Figure 5.4. Instead of the circuit fingerprint, the fingerprint of the HSDir the descriptor is uploaded to is stored in the attribute *fingerprint*.

- **LogEvent**

  With the help of *LogEvents* one gains access to all log events of the Tor process. Each time the process generates a log entry, this event is triggered. The structure of a log event is illustrated in Figure 5.7. The *runlevel* refers to the importance rating of the event, while the actual log message is contained in the field *message*.

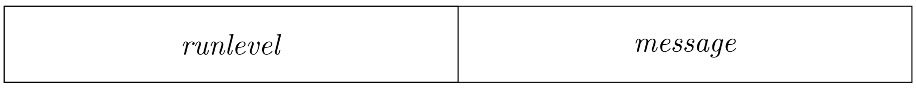


Figure 5.7: Structure of a LogEvent

  Using the log events, the completion of the introduction point establishment can be recognized. For this purpose, all log events are forwarded to the methods *logCallback_v2* or *logCallback_v3* provided in *logAnalyzer.py*, depending on the version of the onion service to be measured. The source code of this file is provided in Annex B.3. The duration of the introduction point establishment is calculated in the analysis using the end time of the introduction circuit creation and the introduction point established notification from the log events. To be able to link these two types of events, the unique circuit id must be extracted from the log event. This is achieved by using the regular expressions shown in Listing 4.

```
INTRO_ESTABLISHED_REGEX_V3 = 'service_handle_intro_established\(\): Successfully received an INTRO_ESTABLISHED cell on circuit ([0-9]{1,}) \(id: ([0-9]{1,})\)'
INTRO_ESTABLISHED_REGEX_V2 = 'rend_service_intro_established\(\): Received INTRO_ESTABLISHED cell on circuit ([0-9]{1,}) \(id: ([0-9]{1,})\)'
```

Listing 4: Introduction Point Establishment Log Event Regular Expressions

  Depending on the service version, a different regular expression is used. In addition to the circuit ids, the time at which the event was received is stored.

In the next step, all results have to be collected from the individual event handlers. Since each event handler is operated in its own thread, a solution for safe inter-thread communication has to be used to exchange the results. For this purpose, synchronized queues are used. As the threads are implicitly created by Stem and are not aware of when the provisioning of an onion service is completed, it is difficult to obtain the results from the queues, because it is not possible to check the content of a queue element without removing it from the queue. For this reason, a workaround was developed in which the queue elements are removed from the queue for checking at periodic intervals and are added to the queue again by the handlers at the same frequency. This procedure is terminated and the Tor event listeners are detached once all required values are available. In order to determine when this is the case, the onion service protocol values from its specification are utilized (e.g. number of introduction points). A timer prevents the measurement process from taking too long due to missing or faulty values. Currently, this time-out is set to 180 seconds.

Finally, the collected results have to be correlated and processed in order to be analysed and stored in an uniform format. This necessary post-processing is individual and depends on the data to be processed. The following processing steps were carried out in the course of this thesis:

- **Descriptor Events:**

  In case of the descriptor analyzer, the individual events have already been divided into three lists based on their appropriate action (*created*, *upload* and *uploaded*), which simplifies their processing. One further advantage lies in the storage of the individual events, which now enables an effective correlation. To obtain the descriptor creation time, the object-method *getTime* is used to receive the creation time stamp of the descriptor event objects in the *created* list. These time values are assigned a label, which allows them to be associated with the respective descriptor. The assignment of upload events to their corresponding uploaded events has proven to be more complex, since both events are stored in separate lists. In the first step, all upload event objects are read from the list of upload events and placed in the dictionary, such that the individual upload procedure can be accessed using a label. Subsequently, the relevant uploaded event is extracted from the list of all uploaded events on the basis of its unique circuit fingerprints. Finally, the timestamps of these uploaded events are obtained using the object method *getTime* and assigned to the upload procedure using its label. In addition, the path of the upload circuit and the directory used for each upload are stored. Thereby, the path of an upload circuit is obtained from circuit events and the responsible directory is obtained by querying the Tor network status using the unique relay fingerprint from the upload events.

- **Circuit Events:**

  To obtain the start and end time of the introduction circuit creation, initially the circuit id of the circuit to be analysed has to be obtained. This is done by firstly parsing the log events that indicate the establishment of an introduction point. These events contain, amongst other information, the circuit ids of all introduction circuits. This circuit ids can then be used as key in the dictionary of introduction circuits to get all circuit elements that correspond to a specific circuit. The temporally first and last element can now be selected from this set of elements, which results in the start and end events of the introduction circuit. Finally, the time stamps of these elements are extracted using the *getTime* method of the circuit element objects.

- **Log Events:**

  The last time values to be obtained are the introduction point establishment times. As mentioned above, the log events are utilized for this purpose. Since these events are not stored within objects, the time stamps are passed in the form of lists, which contain, apart from the timestamp, the ids of the introduction circuits. The time stamps are extracted and saved in a labeled data structure, to be then assigned to their corresponding introduction circuits.

All measurement fields for which no value could be determined due to a timeout or any other error are assigned the value *FAIL* in order to be able to distinguish them from regular fields later in the evaluation. Finally, the measurement results are saved in a csv file using the Python module *csv*, as this format is supported by the most common analysis tools.

5.3.2 Container Environment

As described in section 5.2, the measurement is carried out in a containerized environment to avoid falsification of the results by any influences. The virtualization software Docker is used for this purpose, which uses operating system on-board techniques like *control groups* or *namespaces* to isolate the individual environments [56]. Its architecture consists primarily of three parts. The *Docker Client* communicates with the *Docker Daemon*, which is responsible for the building, running and distribution of Docker containers. All containers are based on an image, which is a template that contains all necessary building blocks for the application that shall be hosted. *Docker Registries* are services that store and provide such images. Besides the predefined images, one can also create customized images, which are either written from scratch or based on other images. All the steps that are necessary to create a custom image are collected in a so-called *Dockerfile*. [56]

Since the requirements for the onion service measurement cannot be fulfilled by a standard environment, a custom image based on *Debian* was created. The most important excerpts from the Dockerfile created for this purpose are described below. The complete file is provided on GitLab [57] and in Annex B.7.

One of the main components of the resulting image is the Tor binary, which is required for the communication with the Tor network. As described in section 5.3.1, in addition to circuit and descriptor events, log events are also required to measure the provisioning time of an onion service. Thereby, the logs contain the notifications of the establishments of introduction points, which are then linked to the corresponding introduction circuit events. This linking takes place on the basis of the unique circuits identifiers. Since circuit events and log events use different storage structures that have different identifiers in the context of introduction circuits, the Tor source code had to be adapted to establish a shared identifier. Specifically, the log outputs of the introduction point establishment notification for V2 services (see Listing 5) and V3 (see Listing 6) services were changed in a way that, in addition to the structure-specific identifier, the global circuit identifier is also provided. This enables the linking of circuit and log events.

```
//rendservice.c:3443
log_debug(LD_REND, "Received INTRO_ESTABLISHED cell on circuit %u (id: %" PRIu32 ")
↪  for service %s", (unsigned)circuit->base_.n_circ_id, circuit->global_identifier,
↪  serviceid);
```

Listing 5: Tor Source Code Log Extensiveness Extension V2

```
//hs_services.h:3324
log_debug(LD_REND, "Successfully received an INTRO_ESTABLISHED cell on circuit %u
↪  (id: %" PRIu32 ") for service
↪  %s",    TO_CIRCUIT(circ)->n_circ_id,circ->global_identifier,
↪  safe_str_client(service->onion_address));
```

Listing 6: Tor Source Code Log Extensiveness Extension V3

The modified Tor source code is added to the container and built using build automation tools. The second important component of the container is the Python measurement environment required to measure the provisioning time of onion services. In order to be able to use it in the container, the Python environment and a package-management system are installed in the first step. The package-management system is used to install the Python modules *Stem* and *Vanguards*. Stem provides the basis for the measurement procedure and is described in section 4.4.1. The Vanguards module adds additional security features to onion service, which are delineated in section 3.2.5. These security features can be additionally adjusted by a configuration file. To obtain comparable results, the default configuration of Vanguards was used in the course of this work. In order to make the analysis module usable, finally its source code is added into the container.

Additionally, in order to automatize the measuring process as far as possible, a script was developed which performs all necessary steps to start the measurement procedure based on the measurement parameters provided. These parameters include the version and type of the onion service to be measured, a timeout which is awaited before the measurement is aborted and the number of introduction points used. The script starts the Tor process using the containers *torrc* file and subsequently starts the Python measurement procedure that generates a csv file with the measurement results. To ensure that this script is invoked when the container is executed, the script is set as the entry point of the Docker image. The source code of the script is provided in Annex B.10.

5.3.3 Analysis Server

If a container based on the containerized measurement environment is executed, one measurement cycle is performed. To obtain representative results, it is necessary to automatically carry out numerous measurements cycles. For this purpose, the analysis server in the form of a bash script was developed, whose source code can be found in Annex B.8.

The following parameters can be set in this script:

- **Target directory:** Path to the directory where the results of the measurements are stored.
- **Docker image:** Name and tag of the custom docker image to be executed.
- **Timeout:** Individual measurement timeout.
- **Introduction points:** Number of introduction points to be expected.

In order to be able to conduct the measurements of the individual versions and types of onion services under comparable conditions, all measurement variants are carried out serially one after the other within one measurement cycle. Finally, the individual files of the measurement cycles are merged with the help of a Python script in order to obtain one single csv file for each measurement type.

5.4 Results

The provisioning times of onion services were measured in the four different variants V2, V3, V3 non-ephemeral and V3 with Vanguards using the implementation described in section 5.3. To obtain the results presented in this section, two measurement sessions were conducted:

1. In the course of the first measurement session, 1520 individual measurements per variant were carried out. This session covered a time period of about three days, which reduces the sensitivity of the measurement results to errors due to temporal fluctuations in the performance of the server or the Tor network.
2. As the first measurement session revealed interesting irregularities, which could not be analysed without additional data material, a second measurement was carried out with an extended measurement scope. In the course of this second session approximately 500 measurements were carried out per variant.

Table 5.2 shows the most important statistical characteristics of the first measurement session. Note that all values in the table are specified in seconds.

Type	*Samples*	*Mean*	*Median*	*SD*	*25%*	*75%*	*Min*	*Max*	*Error*
V2 default	1520	32.82	32.53	3.88	31.83	32.95	26.93	133.53	4.21%
V3 default	1520	5.13	4.23	3.76	3.10	4.23	0.57	93.35	12.69%
V3 non-ephemeral	1520	5.04	4.19	3.85	3.07	6.02	0.79	104.53	14.27%
V3 vanguards	1520	5.49	4.36	5.09	3.25	6.30	1.15	94.08	15.26%

Table 5.2: Total Provisioning Time of Different Onion Service Types

A graphical representation of these numbers in the form of a box plot is provided in Figure 5.8.

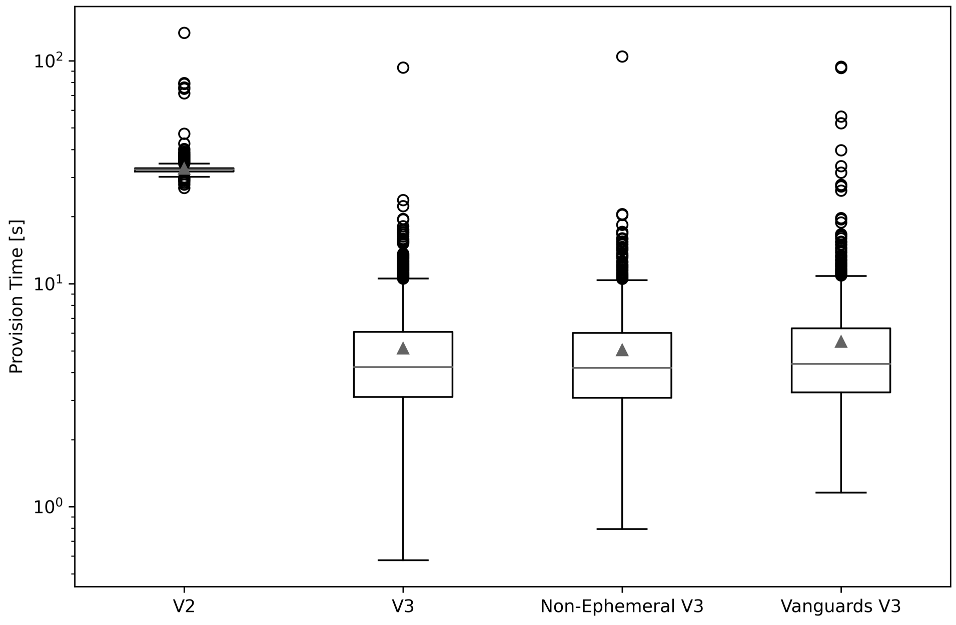


Figure 5.8: Total provisioning Time of different Onion Service Types

The provisioning time of a V2 service averaged at 32.82 seconds, that of V3 services at 5.13 seconds, that of non-ephemeral V3 services at 5.04 seconds and that of V3 services with Vanguards plugin enabled at 5.49 seconds. These numbers indicate that the average provisioning time of an onion service V3 is equivalent to about one sixth of the provisioning time of an onion service V2. Also, the minimum provisioning time of V2 services is significantly higher compared to V3 services. The maximum values of the provisioning time also differ, but not as drastically as the minimum values. The error rate, which indicates whether one of the sub-steps in the provisioning of an onion service has failed, is about two-thirds higher for onion service based on V3 than for V2 services. Since the median and the mean of the individual measurements variants are close to each other, the results appear to be evenly distributed. In order to be able to use statistical tests to compare the provisioning times of the individual variants, it is necessary to ascertain whether the data are normally distributed. In the first step, this assessment is accomplished with the help of a *quantile-quantile* (q-q) diagram, which is shown in Figure 5.9.

The q-q diagram is an efficient way to compare the distributions of two data sets. Thereby, the quantiles of a distribution are plotted against the quantiles of another distribution. If the quantiles of the two distributions are approximately equal, the quantile-quantile points in the diagram should form a straight line, represented by the red line in the graph. In order to check if the measured data is normally distributed, it is plot against generated normally distributed data. [58]

As can be seen in the individual graphs in Figure 5.9, the results of none of the measurement variants are normally distributed, since q-q points significantly deviate from the ideal line. To confirm the

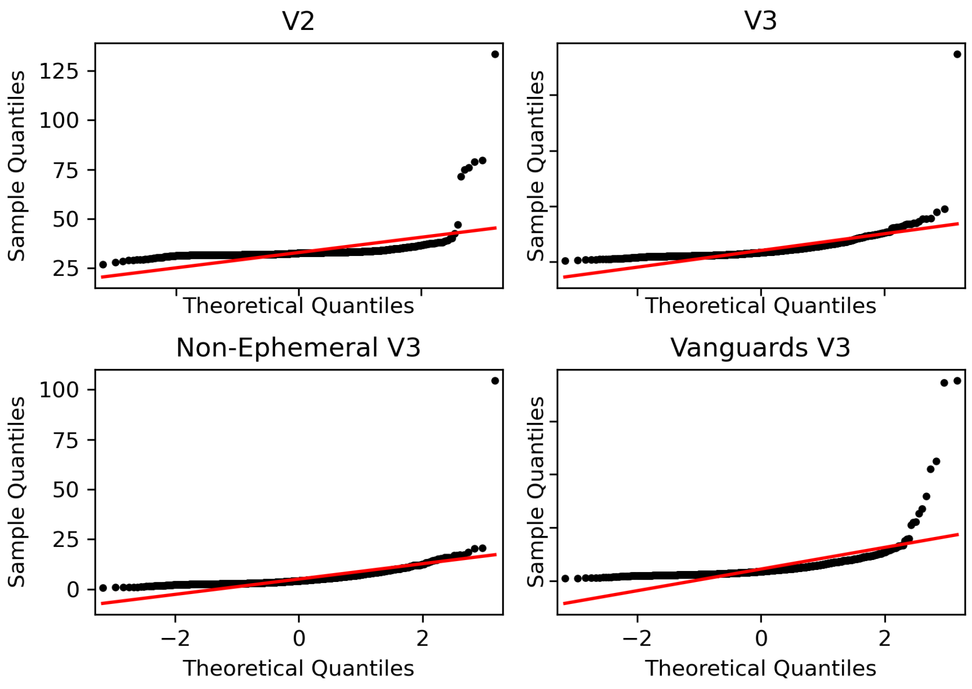


Figure 5.9: Quantile-Quantile (Q-Q) Diagram

graphical results quantitatively, a *Shapiro-Wilk* normality test with a significance level of 1% was carried out for all measurement variants, which also came to the conclusion that the results are not normally distributed. Therefore, the non-parametric *Mann-Whitney U* test is used to compare the provisioning times. In this analysis, the overall provisioning time is the dependent variable and the measurement variable is independent. The following hypotheses are used in the tests:

- H_0: The samples of both measurement variants are from the same distribution. This means that there are no significant deviations in the provisioning times of the two onion service variants considered.
- H_1: The samples of both measurement variants are not from the same distribution. This means that there are significant deviations in the provisioning times of the two onion service variants considered.

The results of the Mann-Whitney U test are shown in Table 5.3.

Type one	*Type two*	*p*	*Result*
V2 default	V3 default	< 0.0001	reject H0
V3 default	V3 non-ephemeral	0.0125	can't reject H0
V3 default	V3 vanguards	0.5255	can't reject H0
V3 non-ephemeral	V3 vanguards	0.0016	reject H0

Table 5.3: Results of the Mann-Whitney U Test

Based on a significance level of 1%, H_0 is rejected if the p value of the Mann-Whitney U test is less than 0.01. Comparing the measured values of the provisioning times of V2 and V3 onion services, the null hypothesis is rejected, which means that the two distributions differ significantly. The same applies to the comparison of non-ephemeral V3 services and V3 services that use the Vanguards plugin. There is no significant difference between V3 onion services and the two onion services V3 based variants. From this results in can be concluded, that there are significant differences in the provisioning times of V2 onion services and V3 onion services while the different variants of the V3 onion services differ only slightly.

The bar chart in Figure 5.10 provides a visual comparison of the duration of the individual sub-steps of the provisioning of the different onion service variants. The mean values of the provisioning sub-steps used for this plot are also provided in Table 5.4. The term *SD* stands for standard deviation and all values in this table are provided in seconds.

Type	*Introduction Point Establishment*	*SD*	*Descriptor Creation*	*SD*	*Upload*	*SD*
V2 default	0.3955	0.3594	30.9890	0.6475	1.4383	3.8020
V3 default	0.4173	0.6143	0.7665	0.2463	3.9725	3.7332
V3 non-ephemeral	0.3585	0.3319	0.8178	0.2775	3.9200	3.8398
V3 vanguards	0.6350	0.3921	0.6220	0.4095	4.2403	5.0283

Table 5.4: Sub-steps of the Provisioning Time of Different Onion Service Types

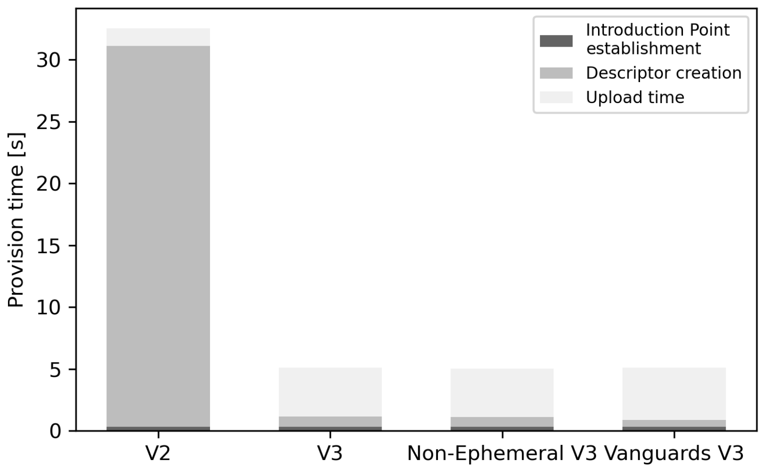


Figure 5.10: Onion Service Provisioning Sub-step Duration

In comparison to the mean duration of the other sub-steps involved in the provisioning of an onion service, the time required for the establishment of introduction points had the least impact on the overall provisioning time. This applies to all four variants, for which the introduction point establishment took approximately the same amount of time. This contrasts with the descriptor creation time that took about 30.98 seconds for onion service V2, which is about 38 times longer than for the other variants. Also the upload time differs significantly between V2 and V3 services. Since the standard deviation of the upload times is very high compared to the standard deviation of the other sub-steps, it can be assumed that the upload times are spread out and not clustered around the median. The distribution of the upload times is shown in the bar chart in Figure 5.11.

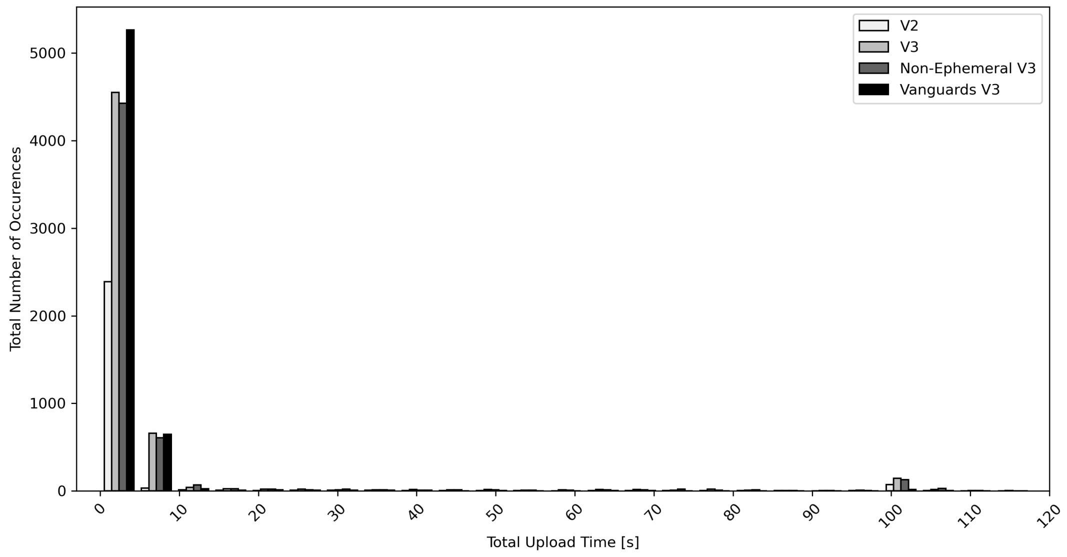


Figure 5.11: Onion Service Upload Time Distribution

From the graph it can be concluded that the majority of uploads took about 5 seconds, while almost all uploads were completed after 20 seconds. An interesting result of the analysis is that some uploads required more than 100 seconds. In order to further investigate this issue, additional measurement values were collected during the second measurement session, which allow the subdivision of the upload time into the duration of the creation of the upload circuits and the actual HTTP upload time. The resulting distributions are depicted in Figure 5.12 and Figure 5.13. From these bar plots it can be concluded that the conspicuously high total upload time is due to issues with the creation of the upload circuits.

5.5 Discussion

The results of the performance measurements provided in section 5.4 indicate that the usage of the Unlinkable Onion Services Protocol (UOSP) in its current version in principle is not feasible with regard to its additional latency. According to the work of Wendolsky et al. [6], the duration inexperienced users of anonymization services are willing to wait is approximately four seconds. Assuming that the data service provisioned in the course of the UOSP is an onion service version

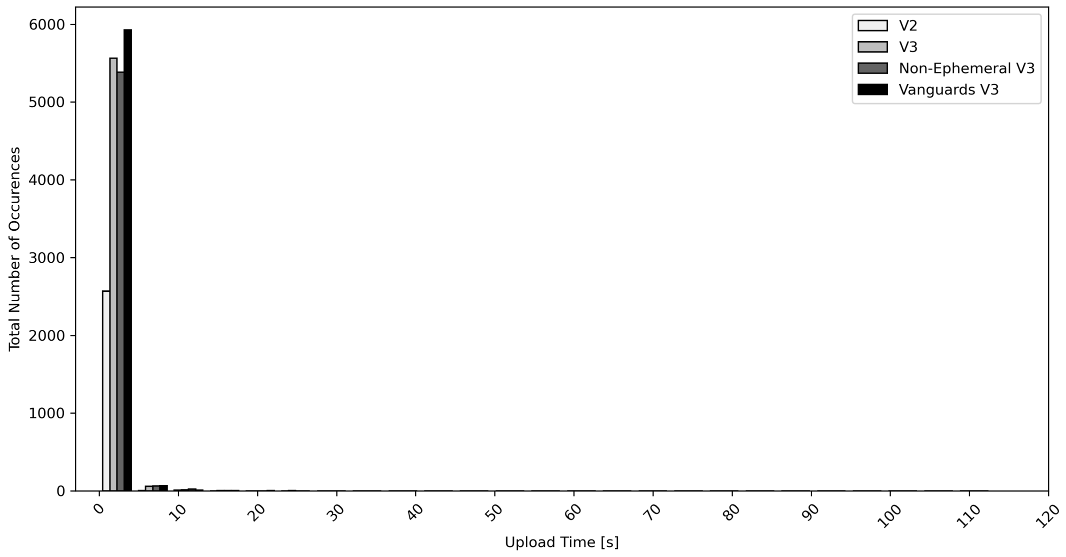


Figure 5.12: Onion Service HTTP Upload Duration

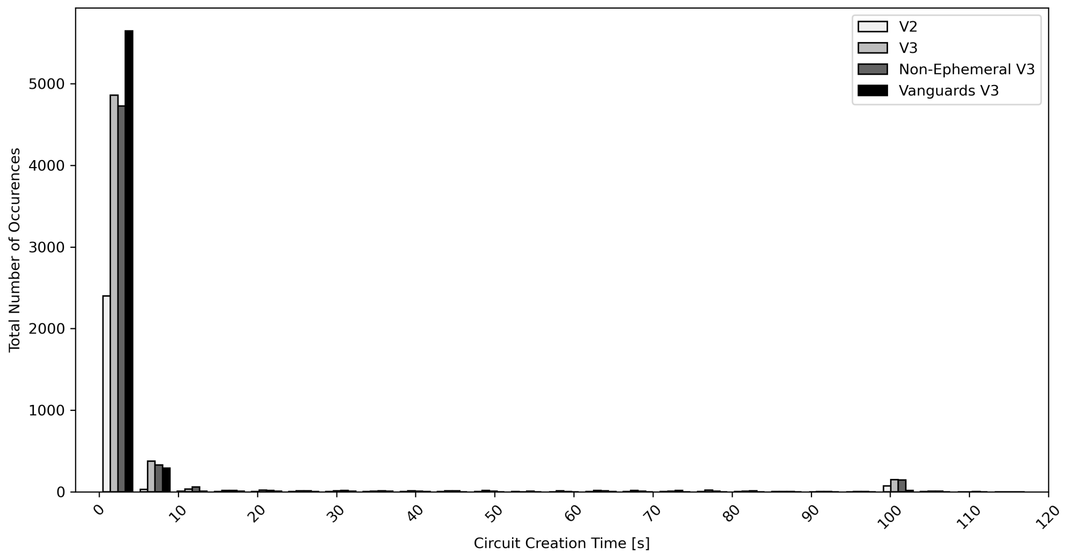


Figure 5.13: Onion Service Upload Circuit Creation Duration

three, this provisioning would require 5.13 seconds on average. In addition to provisioning, the overall response time is composed of the following three HTTP round trips:

1. Client UOSP request to the proxy and its response
2. Proxy UOSP request to the server and its response
3. Proxy UOSP acknowledgement to the server and its response

To be able to approximate the overall response time, HTTP Round Trip Times (RTTs) from the measurement server to an external HTTP server were conducted, whose results are shown in Table 5.5. In total 1000 measurements were performed. The payload of the HTTP responses was kept minimal to simulate UOSP messages.

Mean	*Median*	*SD*	*25%*	*75%*	*Min*	*Max*
0.4097	0.3964	0.0408	0.3845	0.4216	0.3665	0.8124

Table 5.5: HTTP Round Trip Times (RTT)

Based on the average RTT, a client would have to wait approximately 6.35 seconds for the first response from the proxy, which clearly exceeds the four-second limit. In contrast, if a UOSP data service has already been provided and its address is in the cache of the proxy, the overall response time consists solely of an HTTP round trip and the query of the cache of the proxy. This would significantly fall below the four-second limit and thus enable the use of the UOSP.

As can be observed from the data in Table 5.2, the average provisioning of onion service V2 is considerably slower compared to V3 services. After combining this finding with the durations of the provisioning sub-steps in Table 5.4, it becomes apparent that the creation of the service descriptors takes up almost the entire provisioning time. This is particularly remarkable because the generation of V2 descriptors requires much less cryptographic effort than V3 descriptors. The first assumption to explain this discrepancy was that the cryptography used in V2 performed poorly on the measurement system. Since at the time of descriptor creation the RSA key pair of the V2 service is already generated, this cannot have any impact on the duration of the descriptor creation. The cryptographic operations performed during the creation of the descriptors are the hashing of values and the signing of the descriptor. For hashing, the outdated hash algorithm *SHA1* is used, whose execution on the measurement system took place in less than 0.0001 seconds. The signature is an RSA signature that uses the *PKCS.1* padding. Such a signature could be created on the mesasurement system in about 0.59 seconds on average. These results suggest that the cryptographic operations are not decisive for the long creation time of the descriptors of V2 onion service. The real reason for the significantly increased descriptor creation times was finally found in the source code. The first time a descriptor for a V2 onion service is published, a random delay is added, to ensure that the descriptor is stable before it is uploaded. The minimal delay is 30 seconds, which explains the clustering of descriptor creation times at around 31 seconds. Nevertheless, the V2 descriptors required less upload time, which is most likely due to their reduced size compared to the V3 descriptors. [39]

It was anticipated that the provisioning times would not differ drastically between the different V3 services. In principle, this can be confirmed on the basis of the measurement results, but there were significant differences between non-ephemeral onion services and onion services that use the Vanguards plugin. This becomes particularly evident in the comparison of introduction point establishment times, where V3 services that used vanguards required about twice the time of non-ephemeral onion services. This is probably due to the fact that introduction circuits are extended by one node when using the Vanguards plugin. Surprisingly, services using Vanguards built their descriptors faster than the rest of the variants. This seemed initially unreasonable, since there is no difference in how the descriptors are built. In order to provide an explanation for this phenomenon, a number of measurements were performed with different numbers of introduction points. These measurements yielded interesting results, as measurements that used more introduction points took longer to establish their introduction points, but completed the creation of their descriptor at the same time as services that used fewer introduction points. This implies that parts of the descriptors are created in parallel to the establishment of the introduction points, which in turn indicates that the number of introduction points has no effect on the overall provisioning time. As expected, the descriptor upload took the longest with the services that use Vanguards, as the upload circuits also contain an additional hop.

A remarkable finding in the results was the high standard deviation of the upload duration compared to the durations of the other sub-steps. The bar plot in Figure 5.11 revealed that a large proportion of the descriptor uploads are completed within a reasonable time frame (1-10 seconds), whereas several of the descriptor uploads required more than 100 seconds. To determine the cause of this behaviour, a further measurement was conducted that allows the distinction between the upload circuit creation time and the mere HTTP upload time. An analysis of the results of these measurements showed that, contrary to expectations, the upload circuit creation was the main issue. A further source code analysis revealed that there is a 100 second timeout for the creation of circuits to the HSDirs, after which a new connection attempt is made, which is the reason for the delay.

6 Conclusion

This chapter provides a summary of the main findings of this thesis and their implications for Tor onion services. As this is a very broad field of research and the time limitations of a Master's thesis do not allow to investigate all aspects in detail, several open research topics are noted in the future work section.

6.1 Summary

The functional principles of Hidden Service Directories (HSDirs) provide adversaries with the ability to determine when and how often a service is accessed if the attacker is able to control one or more HSDirs of that service. Combining this information with temporal metadata from an additional source (e.g. the log data of a company's network traffic) could allow the identification of the service client or at least its device. As the behaviour of the HSDirs does not in itself constitute a vulnerability, it cannot be assumed that their functional principle will be changed in the foreseeable future. This is a particular threat to those who use the Tor network and Tor onion service to protect their identity against government repression and surveillance.

This thesis aimed to address this problem by developing a protocol that reduces and, if possible, prevents the collection of metadata at the HSDirs. The fundamental problem consists in the fact that a service can be uniquely identified by its onion address and thus tracked by a potential adversary. In order to dissolve the relation between the service and its address, a two-tier addressing concept was developed. This concept allows clients to issue temporal limited individual onion addresses for a service via a publicly known service address referred to as introduction address. An adversary is not able to establish a connection between this individual address and an onion service, provided that the client does not share the address. However, this only applies to service requests issued using the individual onion address. The initial request to the public introduction address constitutes the same threat to anonymity as it would without the protocol. To prove the feasibility of the proposed protocol, a Proof of Concept (PoC) was implemented in Python for Hypertext Transfer Protocol (HTTP) services.

Since the latency of a service is a crucial factor in terms of its usability, a comprehensive performance analysis was carried out, measuring the provisioning time of onion services in different variants, as this is the most time-consuming component of the protocol. The results reveal that the latency involved in issuing an individual onion addresses is approximately 6.35 seconds. According to the previous investigation on latency acceptance of Wendolsky et al. [6], this latency exceeds the acceptable limit, which would lead to the protocol not being used. However, this limitation does not apply to the usage of the issued individual onion addresses, whose latency is significantly below the limit. This implies that the usage of the protocol is in principle feasible, but the commissioning process of the individual onion addresses must be expedited.
The use of the protocol minimizes the ability of collecting metadata on the HSDirs. However, the proposed protocol has two known limitations:

1. Each client must cache the individual addresses issued to it in order to avoid the necessity of making a request to the introduction address every time it accesses an onion service as this would eliminate the advantage of non-linkable addresses. Nevertheless, this has the effect that the cache of the client allows a complete associability of individual addresses to the introduction addresses, which an adversary could use to monitor which services a client accesses. This trade-off is necessary and the risk of compromise is considered low as the cache is under the control of the client in the current protocol proposal.

2. The approach does not scale. As a new onion service must be provisioned for each individual address, the number of addresses is limited. One possible solution to this problem is to no longer provide each user with their individual address, but to randomly select an address from a pool of addresses.

3. The additional latency introduced through the use of UOSP leads to an average Round Trip Time of 6.35 seconds, which exceeds the acceptable latency limit of 4 seconds proposed by Wendolsky et al. [6]. This circumstance is mitigated in practical use, as the caching of the UOSP data addresses on the proxy side considerably reduces this Round Trip Time.

6.2 Future Work

The following research topics regarding the reduction of metadata when accessing onion services remained unprocessed due to temporal limitations of the master thesis:

- **Quantitative Onion Service Anonymity Measurement**

  In order to be able to better compare and classify the improvements of the proposed protocol or any changes to the onion service specification, it is necessary to develop a method to quantify the anonymity of onion services.

- **Scalability Issues**

  The current protocol proposal suffers from scaling issues, as a new onion service must be provisioned for each individual onion address, which limits the number of possible clients and addresses. A possible solution is to no longer issue an individual address to each user, but to randomly choose an already provided address from a pool. For example, this could be implemented with *OnionBalance* [59], which combines introduction points of several services in a super descriptor, which are then randomly selected by the client.

- **Pool Size**

  In case a solution with a pool of onion addresses is considered, the size of this pool and its impact on the anonymity of the clients has to be investigated.

- **Validity Period**

  Currently, the validity period of individual onion addresses is set to 24 hours. In order to determine an ideal period, the impact of the validity period on the anonymity of onion services needs further investigation.

References

[1] Christina Gardner and Donald L. Amoroso. 'Development of an instrument to measure the acceptance of internet technology by consumers'. In: *Proceedings of the Hawaii International Conference on System Sciences* 37.C (2004), pp. 4143–4152. ISSN: 10603425. DOI: 10.1109/hicss.2004.1265623.

[2] Andrew S Tanenbaum and David J Wetherall. *Computer Networks.* 5th. Upper Saddle River, NJ: Prentice Hall Press, 2010. ISBN: 0132126958.

[3] J. Postel. *Transmission Control Protocol.* USA, Sept. 1981. DOI: 10.17487/rfc0793. URL: https://www.rfc-editor.org/info/rfc0793.

[4] Roger Dingledine, Nick Mathewson and Paul Syverson. 'Tor: The second-generation onion router'. In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13.* San Diego, CA: USENIX Association, 2004, p. 21. DOI: 10.5555/1251375.1251396.

[5] Stefan Köpsell. 'Low Latency Anonymous Communication – How Long Are Users Willing to Wait?' In: *Emerging Trends in Information and Communication Security.* Ed. by Günter Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 221–237. ISBN: 978-3-540-34642-5.

[6] Rolf Wendolsky, Dominik Herrmann and Hannes Federrath. 'Performance Comparison of Low-Latency Anonymisation Services from a User Perspective'. In: *Privacy Enhancing Technologies.* Ed. by Nikita Borisov and Philippe Golle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 233–253. ISBN: 978-3-540-75551-7.

[7] Ana Custura et al. *Onionperf.* 2020. URL: https://gitlab.torproject.org/tpo/metrics/onionperf (visited on 03/12/2020).

[8] Ana Custura, Iain Learmonth and Gorry Fairhurst. 'Measuring mobile performance in the tor network with onionperf'. In: *TMA 2019 - Proceedings of the 3rd Network Traffic Measurement and Analysis Conference* (2019), pp. 233–238. DOI: 10.23919/TMA.2019.8784601.

[9] Andriy Panchenko, Lexi Pimenidis and Johannes Renner. 'Performance analysis of anonymous communication channels provided by tor'. In: *ARES 2008 - 3rd International Conference on Availability, Security, and Reliability, Proceedings* (2008), pp. 221–228. DOI: 10.1109/ARES.2008.63.

[10] Karsten Loesing et al. 'Performance measurements and statistics of Tor hidden services'. In: *Proceedings - 2008 International Symposium on Applications and the Internet, SAINT 2008* (2008), pp. 1–7. DOI: 10.1109/SAINT.2008.69.

[11] Christian Wilms. 'Improving the Tor Hidden Service Protocol Aiming at Better Performances'. Master Thesis. Otto-Friedrich Universität Bamberg, 2008.

[12] Christian Wilms. 'Performance Evaluation of Tor Hidden Services'. In: (2007).

[13] Helge Rausch. *Puppetor.* 2013. URL: https://github.com/tsujigiri/puppetor (visited on 05/12/2020).

[14] Jörg Lenhard, Karsten Loesing and Guido Wirtz. 'Performance measurements of tor hidden services in low-bandwidth access networks'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5536 LNCS.March (2009), pp. 324–341. ISSN: 03029743. DOI: `10.1007/978-3-642-01957-9_20`.

[15] David Goulet. *Onion Service version 2 deprecation timeline.* 2020. URL: `https://blog.torproject.org/v2-deprecation-timeline?page=1` (visited on 05/12/2020).

[16] Lasse Øverlier and Paul Syverson. 'Locating hidden servers'. In: *Proceedings - IEEE Symposium on Security and Privacy* 2006 (2006), pp. 100–114. ISSN: 10816011. DOI: `10.1109/SP.2006.24`.

[17] Matthew K. Wright et al. 'The predecessor attack'. In: *ACM Transactions on Information and System Security* 7.4 (2004), pp. 489–522. ISSN: 1094-9224. DOI: `10.1145/1042031.1042032`.

[18] Alex Biryukov, Ivan Pustogarov and Ralf Philipp Weinmann. 'Trawling for tor hidden services: Detection, measurement, deanonymization'. In: *Proceedings - IEEE Symposium on Security and Privacy* (2013), pp. 80–94. ISSN: 10816011. DOI: `10.1109/SP.2013.15`.

[19] Gareth Owen and Nick Savage. 'Empirical analysis of Tor hidden services'. In: *IET Information Security* 10.3 (2016), pp. 113–118. ISSN: 17518717. DOI: `10.1049/iet-ifs.2015.0121`.

[20] Juan A. Elices and Fernando Pérez-González. 'Locating Tor hidden services through an interval-based traffic-correlation attack'. In: *2013 IEEE Conference on Communications and Network Security, CNS 2013* 2 (2013), pp. 385–386. DOI: `10.1109/CNS.2013.6682740`.

[21] Zhen Ling et al. 'Protocol-level hidden server discovery'. In: *Proceedings - IEEE INFOCOM* (2013), pp. 1043–1051. ISSN: 0743166X. DOI: `10.1109/INFCOM.2013.6566894`.

[22] Muqian Chen et al. 'SignalCookie: Discovering Guard Relays of Hidden Services in Parallel'. In: *2019 IEEE Symposium on Computers and Communications (ISCC).* Vol. 2019-June. IEEE, June 2019, pp. 1–7. ISBN: 978-1-7281-2999-0. DOI: `10.1109/ISCC47284.2019.8969639`. URL: `https://ieeexplore.ieee.org/document/8969639/`.

[23] David M. Goldschlag, Michael G. Reed and Paul F. Syverson. 'Hiding routing information'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1174 (1996), pp. 137–150. ISSN: 16113349. DOI: `10.1007/3-540-61996-8_37`.

[24] M Leech et al. *SOCKS Protocol Version 5.* USA, Mar. 1996. DOI: `10.17487/rfc1928`. URL: `https://www.rfc-editor.org/info/rfc1928`.

[25] Cristopher Baines et al. *Tor Rendezvous Specification - Version 3.* 2020. URL: `https://gitweb.torproject.org/torspec.git/tree/rend-spec-v3.txt`.

[26] Daniel J Bernstein. 'Curve25519: New Diffie-Hellman Speed Records'. In: *Public Key Cryptography - PKC 2006.* Ed. by Moti Yung et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33852-9.

[27] Tim Wilson-Brown et al. *Proposal 260 - Rendezvous Single Onion Services.* 2017. URL: `https://github.com/torproject/torspec/blob/master/proposals/260-rend-single-onion.txt`.

[28] Nick Mathewson. *Proposal 220 - Migrate server identity keys to Ed25519.* 2013. URL: https://gitweb.torproject.org/torspec.git/plain/proposals/220-ecc-id-keys.txt.

[29] Roger Dingledine and Nick Mathewson. *Tor Protocol Specification.* 2020. URL: https://gitweb.torproject.org/torspec.git/plain/tor-spec.txt.

[30] Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA).* Tech. rep. Jan. 2017. DOI: 10.17487/RFC8032. URL: https://tools.ietf.org/html/rfc8032https://www.rfc-editor.org/info/rfc8032.

[31] Aaron Johnson et al. 'Users get routed: Traffic correlation on Tor by realistic adversaries'. In: *Proceedings of the ACM Conference on Computer and Communications Security* (2013), pp. 337–348. ISSN: 15437221. DOI: 10.1145/2508859.2516651.

[32] M Wright et al. 'Defending anonymous communications against passive logging attacks'. In: *2003 Symposium on Security and Privacy, 2003.* 2003, pp. 28–41. DOI: 10.1109/SECPRI.2003.1199325.

[33] George Kadianakis et al. *Tor directory protocol.* 2021. URL: https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt.

[34] Isis Lovecruft et al. *Tor Guard Specification.* 2020. URL: https://gitweb.torproject.org/torspec.git/plain/guard-spec.txt.

[35] George Kadianakis. *Announcing the Vanguards Add-On for Onion Services.* 2018. URL: https://blog.torproject.org/announcing-vanguards-add-onion-services (visited on 13/01/2021).

[36] George Kadianakis and Mike Perry. *Mesh-based vanguards.* 2021. URL: https://gitweb.torproject.org/torspec.git/commit/proposals/292-mesh-vanguards.txt.

[37] Andreas Pfitzmann and Marit Hansen. *A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management.* Aug. 2010. URL: http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf.

[38] Mike Perry. *Tor's Open Research Topics: 2018 Edition.* 2018. URL: https://blog.torproject.org/tors-open-research-topics-2018-edition (visited on 11/12/2020).

[39] Roger Dingledine, Nick Mathewson and Paul Syverson. *Tor Rendezvous Specification - Version 2.* 2019. URL: https://gitweb.torproject.org/torspec.git/tree/rend-spec-v2.txt.

[40] Alec Muffett. *1 Million People use Facebook over Tor.* 2016. URL: https://www.facebook.com/notes/facebook-over-tor/1-million-people-use-facebook-over-tor/865624066877648/ (visited on 17/12/2020).

[41] Tor Metrics. *Unique .onion addresses (version 2 only).* 2020. URL: https://metrics.torproject.org/hidserv-dir-onions-seen.csv (visited on 17/12/2020).

[42] Andy Yen. *ProtonMail.* 2020. URL: https://protonmail.com (visited on 19/12/2020).

[43] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3.* USA, Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[44] Gerard J Holzmann and William Slattery Lieberman. *Design and Validation of Computer Protocols.* Vol. 94. New Jersey: Prentice hall Englewood Cliffs, 1991, p. 500. ISBN: 0135399254.

[45] Alex Biryukov et al. 'Content and popularity analysis of tor hidden services'. In: *Proceedings - International Conference on Distributed Computing Systems* 30-June-20 (2014), pp. 188–193. DOI: 10.1109/ICDCSW.2014.20. arXiv: 1308.6768.

[46] Damian Johnson. *Stem.* 2020. URL: https://stem.torproject.org/ (visited on 01/01/2021).

[47] Doug Hellmann. *SocketServer - Creating network servers.* 2010. URL: https://bip.weizmann.ac.il/course/python/PyMOTW/PyMOTW/docs/SocketServer/index.html (visited on 01/01/2021).

[48] Python Software Foundation. *socketserver - A framework for network servers.* 2021. URL: https://docs.python.org/3/library/socketserver.html (visited on 02/01/2021).

[49] Zope Foundation. *ZODB - a native object database for Python.* 2016. URL: http://www.zodb.org/en/latest/ (visited on 02/01/2021).

[50] Tal Yarkoni. *Transitions.* 2020. URL: https://github.com/pytransitions/transitions (visited on 02/01/2021).

[51] Thomas Raab. *Prototype Unlinkability.* 2021. URL: https://git.ins.jku.at/proj/digidow/prototype-unlinkability (visited on 13/02/2021).

[52] Benoit Chesneau. *http-parser.* 2020. URL: https://github.com/benoitc/http-parser/ (visited on 03/01/2021).

[53] Tor Project. *Configuration file for a typical Tor user.* 2019. URL: https://raw.githubusercontent.com/torproject/tor/master/src/config/torrc.sample.in (visited on 04/01/2021).

[54] Python Software Foundation. *http.server - HTTP servers.* 2021. URL: https://docs.python.org/3/library/http.server.html (visited on 05/01/2021).

[55] Thomas Raab. *Prototype Timing-Analysis.* 2021. URL: https://git.ins.jku.at/proj/digidow/prototype-timing-analysis (visited on 13/02/2021).

[56] Docker Inc. *Docker overview.* 2020. URL: https://docs.docker.com/get-started/overview/ (visited on 17/01/2021).

[57] Thomas Raab. *tor-docker.* 2021. URL: https://git.ins.jku.at/students/tor-docker (visited on 14/02/2021).

[58] John M Chambers et al. *Graphical Methods for Data Analysis.* London: Chapman and Hall/CRC, Jan. 2018. ISBN: 9781351072304. DOI: 10.1201/9781351072304.

[59] George Kadianakis. *Cooking with Onions: Finding the Onionbalance.* 2016. URL: https://blog.torproject.org/cooking-onions-finding-onionbalance (visited on 07/02/2021).

Appendix A: Unlinkable Onion Service Protocol

A.1 Database Utils

```python
import persistent
import persistent.list
from datetime import datetime, timezone
import pytz
import logging
from ZODB import DB, FileStorage

utc = pytz.UTC

TIME_FORMAT = '%Y-%m-%d %H:%M:%S'

"""
Class for connecting to the ZODB database file
"""


class MyZODB(object):
    def __init__(self, path):
        self.storage = FileStorage.FileStorage(path)
        self.db = DB(self.storage)
        self.connection = self.db.open()
        self.dbroot = self.connection.root()

    def close(self):
        self.connection.close()
        self.db.close()
        self.storage.close()


"""
Data representation for storing and managing custom UOSP data addresses on the server
↪ side
"""


class UospDataAddresses(persistent.Persistent):
    def __init__(self):
        self.__uosp_data_addresses = persistent.mapping.PersistentMapping()

    """
    Add a custom UOSP data address to the servers database

    :param uosp_data_address: Custom UOSP data address to be added to the servers
↪   database
    :param uosp_validity: Validity of the newly provisioned address
    """

    def add_data_address(self, uosp_data_address, uosp_validity):
        self.__uosp_data_addresses[uosp_data_address] = uosp_validity
        logging.info(f"Added data address {uosp_data_address}")

    """
    Remove a custom UOSP data address from the servers database

    :param uosp_data_address: Custom UOSP data address to be removed from the servers database
    """

    def remove_data_address(self, uosp_data_address):
        if uosp_data_address in self.__uosp_data_addresses:
            del self.__uosp_data_addresses[uosp_data_address]
            logging.info(f"Removed data address {uosp_data_address}")

    """
    Check the validity of a custom UOSP data address

    :param uosp_data_address: Custom UOSP data address whose validity shall be checked
    :return: True, iff the custom address provided is still valid, false otherwise
    """

    def __check_validity(self, uosp_data_address):
        if uosp_data_address in self.__uosp_data_addresses:
            # get time stamp
            uosp_validity = self.__uosp_data_addresses[uosp_data_address]
            # compare time stamps
            if datetime.now(timezone.utc) > uosp_validity:
                return False
            else:
                return True

    """
    Revise the validity of all currently provisioned custom UOSP addresses

    :param controller: Handle for the stem tor controller
    """

    def revise_validity(self, controller):
        for uosp_address in self.__uosp_data_addresses.copy():
            if not self.check_validity(uosp_address):
                # remove invalid entries from the database
                self.remove_data_address(uosp_address)
                # discontinue invalid services (tear them down)
                controller.remove_ephemeral_hidden_service(uosp_address)


"""
Data representation for storing and managing custom UOSP data addresses on the client side
"""


class UospIntroAddress(persistent.Persistent):

    def __init__(self, uosp_intro_address):
        self.__uosp_intro_address = uosp_intro_address
        self.__uosp_data_addresses = persistent.list.PersistentList()

    """
    Add a custom UOSP data address to the proxy's database

    :param uosp_data_address: Custom UOSP data address to be added to the proxy's database
    :param uosp_validity: Validity of the newly provisioned address
    """

    def add_data_address(self, uosp_data_address, uosp_validity):
        self.__uosp_data_addresses.append([uosp_data_address, uosp_validity])
        logging.info(f"Added data address {uosp_data_address} for intro addresss {self.__uosp_intro_address}")

    """
       Remove a custom UOSP data address from the proxy's database

       :param uosp_data_address: Custom UOSP data address to be removed from the proxy's database
       """

    def remove_data_address(self, uosp_data_address):
        for address_entry in self.__uosp_data_addresses:
            if uosp_data_address == address_entry[0]:
                self.__uosp_data_addresses.remove(address_entry)
                logging.info(f"Removed data address {uosp_data_address} for intro addresss {self.__uosp_intro_address}")

    """
    Revise the validity of all currently provisioned custom UOSP addresses of the objects intro address
    """

    def revise_validity(self):
        for uosp_address in self.__uosp_data_addresses:
            if not self.__check_validity(uosp_address[0]):
                self.remove_data_address(uosp_address[0])

    """
    Get the index of a custom UOSP address in the list of addresses for the objects intro address

    :param uosp_data_address: Custom UOSP address
    :return: Index of the entry if available, None otherwise
    """

    def __get_address_index(self, uosp_data_address):
        for uosp_address in self.__uosp_data_addresses:
            if uosp_address[0] == uosp_data_address:
                return self.__uosp_data_addresses.index(uosp_address)
        return None

    """
    Check the validity of a custom UOSP data address

    :param uosp_data_address: Custom UOSP data address whose validity shall be checked
    :return: True, iff the custom address provided is still valid, false otherwise
    """

    def __check_validity(self, uosp_data_address):
        index = self.__get_address_index(uosp_data_address)
        if index is not None:
            uosp_validity = datetime.strptime(self.__uosp_data_addresses[index][1], TIME_FORMAT)
            uosp_validity = utc.localize(uosp_validity)
            # compare time stamps+
            if datetime.now(timezone.utc) > uosp_validity:
                return False
            else:
                return True

    """
    Revise the validity of all custom UOSP addresses of the objects intro address and
    ↪ return the latest valid entry

    :return: Latest valid custom UOSP address for the objects intro address
    """

    def get_latest_valid_data_address(self):
        self.revise_validity()
        if len(self.__uosp_data_addresses) > 0:
            # return last = latest entry
            return self.__uosp_data_addresses[-1]

    """
    Return the count of custom UOSP addresses for the objects into address
    """

    def get_data_address_count(self):
        return len(self.__uosp_data_addresses)
```

A.2 Connection Utils

```python
import sys
import stem.control
import stem.connection
import logging

"""
Establish a connection to the control port of tor and authenticate
"""
CONTROL_PORT = 9051
CONTROL_ADDRESS = '127.0.0.1'

logging.basicConfig(level=logging.INFO)
logging.getLogger("stem").setLevel(logging.ERROR)


def establish_connection(address=CONTROL_ADDRESS, port=CONTROL_PORT, password=None):
    # get connection to tor via the control port
    try:
        # the explicit declaration of the port isn't necessary since it's the default
        ↪ port
        controller = stem.control.Controller.from_port(address, port)
    except stem.SocketError as exc:
        logging.error(f"Unable to connect to port control port 9051 ({exc})")
        sys.exit(1)

    if controller:
        # authenticate
        try:
            if password != None:
                controller.authenticate(password=password)
            else:
                controller.authenticate()
            logging.info("Connection established and authenticated")
            return controller
        except stem.connection.PasswordAuthFailed:
            logging.error("Unable to authenticate, password is incorrect")
            sys.exit(1)
        except stem.connection.AuthenticationFailure as exc:
            logging.error(f" * Unable to authenticate: {exc}")
            sys.exit(1)


"""
Create new ephemeral Hidden Service and return its onion id
"""


def create_ephemeral_hidden_service(controller, source_port, target_port, version):
    try:
        if version == 2:
            hs = controller.create_ephemeral_hidden_service({source_port:
             ↪  target_port}, await_publication=False,
                                                              key_type="NEW",
                                                               ↪  key_content="RSA1024",
                                                               ↪  detached=True)
        else:
            hs = controller.create_ephemeral_hidden_service({source_port:
             ↪  target_port}, await_publication=False,
                                                              detached=True,
                                                               ↪  key_type='NEW',
                                                               ↪  key_content='BEST')
        if hs.service_id:
            return hs.service_id
        else:
            return None

    except stem.ControllerError as exc:
        logging.error(f"Unable to create the hidden service ({exc})")
        sys.exit(1)


def create_hidden_service(controller, path, port):
    try:
        controller.create_hidden_service(path, port, target_address='127.0.0.1')
    except stem.ControllerError as exc:
        logging.error(f"Unable to create the hidden service ({exc})")
        sys.exit(1)
```

A.3 Server State Machine

```python
from transitions import Machine
import logging


class UOSPServerStateMachine(object):
    states = ['initial_state', 'client_req_received', 'legacy_http_resp_sent',
     ↪  'uosp_resp_sent', 'data_resp_sent',
              'uosp_ack_received', 'server_fin_sent', 'ERROR']

    # Functions

    def __init__(self):
        logging.getLogger('transitions').setLevel(logging.ERROR)

        # Initialize the state machine
        self.machine = Machine(model=self, states=UOSPServerStateMachine.states,
        ↪  initial='initial_state')

        # Transitions

        # s0 ->s1
        self.machine.add_transition('receive_request', source='initial_state',
        ↪  dest='client_req_received')

        # s1 -> s2
        self.machine.add_transition('send_legacy_response',
        ↪  source='client_req_received', dest='legacy_http_resp_sent')

        # s1 -> s3
        self.machine.add_transition('send_uosp_response',
        ↪  source='client_req_received', dest='uosp_resp_sent')

        # s1 -> s4
        self.machine.add_transition('send_data_response',
        ↪  source='client_req_received', dest='data_resp_sent')

        # s3 -> s5
        self.machine.add_transition('receive_uosp_ack', source='uosp_resp_sent',
        ↪  dest='uosp_ack_received')

        # s5 -> ERR
        self.machine.add_transition('pub_key_malformed', source='uosp_ack_received',
        ↪  dest='ERROR')

        # s5 -> s6
        self.machine.add_transition('send_server_fin', source='uosp_ack_received',
        ↪  dest='server_fin_sent')
```

A.4 Server Module

```python
from stem.control import EventType, Controller
from utils.connect import establish_connection, create_ephemeral_hidden_service
from http.server import BaseHTTPRequestHandler, HTTPServer
from socketserver import ForkingMixIn
import time
import hashlib
import uosp_state_machine_server
import logging
import signal
import sys
from datetime import datetime, timezone, timedelta
from utils.db_util import UospDataAddresses, MyZODB
import transaction
import threading
import ssl

# password for control port authentication
```

```python
PASSWORD = 'tor'
# uosp version
UOSP_VERSION = '0.1'
VALIDITY_MINUTES = 1440
VALIDITY_REVISION_MINUTES = 10

DB_PATH = 'data_addresses.fs'

# connect and authenticate
intro_addresses = {}
controller = None


class ForkedHTTPServer(ForkingMixIn, HTTPServer):
    """Handle requests in a separate process"""


class RequestHandler(BaseHTTPRequestHandler):
    uosp_address = None
    uosp_validity = None
    close_connection = False
    controller = None

    # server state machine
    ssm = uosp_state_machine_server.UOSPServerStateMachine()

    def do_GET(self):
        # trigger for uosp data protocol header
        uosp_data = False

        # state changed from initial to request received
        state = self.ssm.state
        self.ssm.receive_request()
        logging.info('State changed from %s to %s' % (state, self.ssm.state))

        # don't close connection after first response
        self.close_connection = False

        # check if UOSP header is set and version is supported
        url = self.headers.get('Host').split('.')[0]
        state = self.ssm.state
        if self.headers.get('UOSP') and self.headers.get('UOSP_VERSION') == '0.1':
            if url in intro_addresses:
                # UOSP
                if self.controller is None:
                    controller = establish_connection(password=PASSWORD)

                message = create_ephemeral_hidden_service(controller, 80, 80, 3)
                self.uosp_validity = datetime.now(timezone.utc) +
                ↪  timedelta(minutes=VALIDITY_MINUTES)
                self.uosp_address = message

                db = MyZODB(DB_PATH)
                # no db entry yet
                if len(db.dbroot) == 0:
                    db.dbroot['addresses'] = UospDataAddresses()
                db.dbroot['addresses'].add_data_address(self.uosp_address,
                ↪  self.uosp_validity)
                transaction.commit()
                db.close()

                self.ssm.send_uosp_response()
            else:
                # UOSP, but no into URL -> deliver web page
                self.ssm.send_data_response()
                message = "<HTML><HEAD><TITLE>Pretty Web
                ↪  Page</TITLE></HEAD><BODY><h3>Web Page</h3>This is the web " \
                          "page the client intended to reach</BODY></HTML>"
                uosp_data = True

        else:
            # No UOSP, deliver legacy page
            self.ssm.send_legacy_response()
            message = "<HTML><HEAD><TITLE>UOSP
            ↪  Response</TITLE></HEAD><BODY><h3>UOSP</h3>This page supports the " \
                      "Unlinkable Onion Service Protocl (UOSP). When you receive this
                      ↪  message, your client/browser " \
                      "doesn't support this protocol. </BODY></HTML>"
            uosp_data = True

        logging.info('State changed from %s to %s' % (state, self.ssm.state))

        self.protocol_version = "HTTP/1.1"
        self.send_response(200)
        self.send_header("Content-Length", str(len(message)))
        self.send_header("UOSP_DATA", str(uosp_data))
        self.send_header("UOSP", 'True')
        self.send_header("UOSP_VERSION", UOSP_VERSION)

        # add validity if intro url used
        if self.uosp_validity is not None:
            self.send_header("UOSP_VALIDITY", self.uosp_validity.strftime("%Y-%m-%d
            ↪  %H:%M:%S"))
        self.end_headers()
        self.wfile.write(bytes(message, "utf8"))
        return

    def do_POST(self):
        self.close_connection = True
        content_len = int(self.headers.get('Content-Length'))
        post_body = self.rfile.read(content_len).decode()

        state = self.ssm.state
        self.ssm.receive_uosp_ack()
        logging.info('State changed from %s to %s' % (state, self.ssm.state))

        # calculate hash value
        hash_pub = hashlib.sha256()
        hash_pub.update(self.uosp_address.encode())
        hash_pub = hash_pub.hexdigest()

        state = self.ssm.state
        if str(hash_pub) == post_body:
            message = "FIN"
            self.ssm.send_server_fin()
        else:
            self.ssm.pub_key_malformed()
            message = "ERROR"

        logging.info('State changed from %s to %s' % (state, self.ssm.state))
```

```python

        self.protocol_version = "HTTP/1.1"
        self.send_response(200)
        self.send_header("Content-Length", str(len(message)))
        self.send_header("UOSP", 'True')
        self.send_header("UOSP_DATA", 'FALSE')
        self.send_header("UOSP_VERSION", UOSP_VERSION)
        self.end_headers()
        self.wfile.write(bytes(message, "utf8"))
        return


"""
Called when the service shuts down to delete all created ephemeral onion services
"""


def exit_gracefully(signum, frame):
    service_list = controller.list_ephemeral_hidden_services(default=None,
    ↪   detached=True)
    for service in service_list:
        controller.remove_ephemeral_hidden_service(service)
        logging.info(f"Service with uri {str(service)} deleted")
    logging.info("Shutting down server")
    sys.exit(0)


def revise_uoasp_data_addesses(ctrl):
    while True:
        time.sleep(VALIDITY_REVISION_MINUTES * 60)
        db = MyZODB(DB_PATH)
        if 'addresses' in db.dbroot:
            db.dbroot['addresses'].revise_validity(ctrl)
        logging.info("Revised UOSP data addresses")
        transaction.commit()
        db.close()


if __name__ == '__main__':
    controller = establish_connection(password=PASSWORD)
    service = create_ephemeral_hidden_service(controller, 80, 80, 3)
    intro_addresses[str(service)] = datetime.now(timezone.utc)
    logging.info(f"Service with uri {str(service)} created")
    # start thread that periodically revises the validity of provisioned custom onion
    ↪   services
    threading.Thread(target=revise_uoasp_data_addesses, args=(controller,),
    ↪   daemon=True).start()
    ForkedHTTPServer.allow_reuse_address = True
    signal.signal(signal.SIGINT, exit_gracefully)
    server = ForkedHTTPServer(('localhost', 80), RequestHandler)
    server.socket = ssl.wrap_socket(
        server.socket,
        keyfile="key.pem",
        certfile='cert.pem',
        server_side=True)
    server.serve_forever()
```

A.5 Server Tor Configuration

```
## Logging
Log notice file /var/log/tor/notices.log
Log info file /var/log/tor/info.log
Log debug file /var/log/tor/debug.log

## Control port
ControlPort 9051
HashedControlPassword 16:0717EF71CDF228DE60E93242A4AFE681FB525080DC1A330A103FAD9782

## Run as daemon in background
RunAsDaemon 1
```

A.6 Proxy State Machine

```
from transitions import Machine
import logging

class UOSPClientStateMachine(object):
    states = ['initial_state', 'cache_addr_used', 'client_req_sent',
    ↪  'server_pub_key_received', 'client_pub_ack_sent',
              'server_fin_received', 'data_resp_received', 'ERROR']

    # Functions
    def __init__(self):
        logging.getLogger('transitions').setLevel(logging.ERROR)

        # Initialize the state machine
        self.machine = Machine(model=self, states=UOSPClientStateMachine.states,
        ↪  initial='initial_state')

        # Transitions

        # s0 ->s1
        self.machine.add_transition('send_request', source='initial_state',
        ↪  dest='client_req_sent')

        # s0 -> s6
        self.machine.add_transition('use_cache_addr', source='initial_state',
        ↪  dest='cache_addr_used')

        # s1 -> s2
        self.machine.add_transition('receive_pub_key', source='client_req_sent',
        ↪  dest='server_pub_key_received')

        # s2 -> s3
        self.machine.add_transition('send_uosp_ack',
        ↪  source='server_pub_key_received', dest='client_pub_ack_sent')

        # s2 -> ERR
        self.machine.add_transition('pub_key_malformed',
        ↪  source='server_pub_key_received', dest='ERROR')

        # s3 -> ERR
        self.machine.add_transition('server_hash_invalid',
        ↪  source='client_pub_ack_sent', dest='ERROR')

        # s3 -> s4
        self.machine.add_transition('receive_server_fin',
        ↪  source='client_pub_ack_sent', dest='server_fin_received')

        # s1 -> s5
        self.machine.add_transition('receive_data', source='client_req_sent',
        ↪  dest='data_resp_received')
```

A.7 Proxy Module

```
import logging
import select
import socket
from socketserver import ForkingMixIn, TCPServer, StreamRequestHandler
import tldextract
import socks
import validators
from utils.db_util import UospIntroAddress, MyZODB
import transaction

import uosp_state_machine_client
import hashlib

import ssl

# try to import C parser then fallback in pure python parser.
try:
    from http_parser.parser import HttpParser
except ImportError:
    from http_parser.pyparser import HttpParser

logging.basicConfig(level=logging.INFO)

# running tor instance
UPSTREAM_HOST = '127.0.0.1'
UPSTREAM_PORT = 9050

# proxy configuration
PROXY_HOST = '0.0.0.0'
PROXY_PORT = 9011

UOSP_VERSION = '0.1'

DB_PATH = 'intro_addresses.fs'


# base class for threaded TCP server
class ForkingTCPServer(ForkingMixIn, TCPServer):
    pass


"""
For every new client connection, a new thread is started , which instantiates the
↪  SocksProxy class as handler for its requests
"""


class SocksProxy(StreamRequestHandler):
```

```python
    # client state machine
    csm = uosp_state_machine_client.UOSPClientStateMachine()

    # placeholder variables which are used to store the received intro host and temporary uosp address received from the server
    intro_host = None
    uosp_address = None
    uosp_validity = None
    legacy_request = False
    cache_request = False

    """
    Handle state changes

    :param data Data received on a socket connection (can be request/response)
    :param recv_socket Handle for the socket used to send a response to the remote server (avoid separate socket connections for each message pair)
    """

    def handle_state_change(self, data, recv_socket=None):
        # print(self.uosp_data_addresses)
        # get current state
        state = self.csm.state

        # transition from initial state when the client request is received
        if state == 'initial_state':
            if not self.cache_request:
                self.csm.send_request()
            else:
                self.csm.use_cache_addr()

            logging.info('State changed from %s to %s' % (state, self.csm.state))

        # transition from client_req sent to server_pub_key_received
        if state == 'client_req_sent':
            if data[0]:
                # this is a uosp data response
                self.csm.receive_data()
            else:
                self.csm.receive_pub_key()
                logging.info('State changed from %s to %s' % (state, self.csm.state))

                # store received onion address for later use
                self.uosp_address = data[2]

                # the pub key was received, send its hash value back to the server to ensure error-free transmission
                if self.intro_host is not None:
                    # calculate hash value
                    hash_pub = hashlib.sha256()
                    hash_pub.update(self.uosp_address.encode())
                    hash_pub = hash_pub.hexdigest()

                    request_string = 'POST /session HTTP/1.1\r\nHost:' + self.intro_host + '\r\nContent-Type: text/plain\r\nContent-Length: ' + str(
                        len(hash_pub)) + '\r\nUOSP: True\r\nUOSP_VERSION:' + UOSP_VERSION + '\r\n\r\n' + hash_pub

                    # use the socket handle passed within the parameters to send the response (avoid new socket connection for every message pair)
                    recv_socket.sendall(request_string.encode())
                    self.csm.send_uosp_ack()
                    logging.info('State changed from %s to %s' % (state, self.csm.state))

        # transition from client_pub_ack_sent to server_fin_received or ERROR
        if state == 'client_pub_ack_sent':
            if data[2] == 'FIN':
                self.csm.receive_server_fin()
                logging.info('State changed from %s to %s' % (state, self.csm.state))
            else:
                self.csm.server_hash_invalid()
                logging.info('State changed from %s to %s' % (state, self.csm.state))

    """
    Receive HTTP request and parse its header

    :return Dictionary containing the decoded relevant header fields
    """

    def parse_http_header(self):
        p = HttpParser()
        header = {}

        # read data from socket
        data = self.connection.recv(1024)

        recved = len(data)
        nparsed = p.execute(data, recved)
        assert nparsed == recved

        # header completely received
        if p.is_headers_complete():
            http_header = p.get_headers()

            # extract host address
            header['host'] = http_header['Host'].split(':')[0]

            # extract suffix from host (must be  a public ICANN TLD)
            no_fetch_extract = tldextract.TLDExtract(suffix_list_urls=None)
            header['suffix'] = no_fetch_extract(http_header['host']).suffix

            # extract host port
            if len(http_header['Host'].split(':')) > 1:
                header['port'] = http_header['Host'].split(':')[1]
            else:
                header['port'] = 80

        if len(header) > 0:
            return header
        else:
            return None

    """
    Parse the body of a HTTP response

    :param data The data is read from the socket, alternatively the already read HTTP
    message can be passed
    :return List containing information whether its an uosp data response and the
↪   parsed decoded http body
    """

    def parse_http_body(self, sock_conn):
        p = HttpParser()
        body = b""
        uosp_data = False
        uosp = False

        while True:
            data = sock_conn.recv(1024)
            if not data:
                break

            recved = len(data)
            nparsed = p.execute(data, recved)
            assert nparsed == recved

            if p.is_headers_complete():
                headers = p.get_headers()
                # check if uosp data response
                if 'UOSP' in headers:
                    uosp = True

                if 'UOSP_DATA' in headers:
                    if headers['UOSP_DATA'] == 'True':
                        uosp_data = True
                if 'UOSP_VALIDITY' in headers:
                    self.uosp_validity = headers['UOSP_VALIDITY']

            if p.is_partial_body():
                body += (p.recv_body())

            if p.is_message_complete():
                break

        return [uosp_data, uosp, body.decode().rstrip()]

    """
    Function called when a new request is received
    """

    def handle(self):
        logging.info('Accepting connection from %s:%s' % self.client_address)

        # parse http header of the received request to extract host and port
        http_header = self.parse_http_header()

        if http_header is not None:
            # only onion traffic triggers the uosp, normal http traffic is proxied
            ↪  normally
            if http_header['port'] == 80 and http_header['suffix'] == 'onion':
                # store the initial host and port for later use (UOSP ACK)
                self.intro_host = str(http_header['host']) + ':' +
                ↪  str(http_header['port'])

                # check if uosp data adress is already in the cache db
                db = MyZODB(DB_PATH)

                if self.intro_host.split('.')[0] in db.dbroot:
                    print(db.dbroot[self.intro_host.split('.')[0]])
                    db.dbroot[self.intro_host.split('.')[0]].revise_validity()

                if self.intro_host.split('.')[0] not in db.dbroot or db.dbroot[self.intro_host.split('.')[0]].get_data_address_count() < 1:
                    logging.info("No cached uosp data address for intro address " + self.intro_host.split('.')[0])
                    # create socket connection to requested resource using the upstream socks proxy
                    remote = socks.socksocket()
                    remote.set_proxy(socks.SOCKS5, UPSTREAM_HOST, UPSTREAM_PORT)  # SOCKS4 and SOCKS5 use port 1080 by default
                else:
                    cache_address = db.dbroot[self.intro_host.split('.')[0]].get_latest_valid_data_address()
                    self.cache_request = True
                    self.handle_state_change(None)
                    message = "<HTML><HEAD><TITLE>UOSP Response</TITLE></HEAD><BODY><h3>UOSP</h3>This page supports the " \
                              "Unlinkable Onion Service Protocl (UOSP). Please use the following link <a href='http://" + cache_address[0] + ".onion'> USOP Link</a></BODY></HTML>"

                    data = 'HTTP/1.1 200 OK\r\nServer: BaseHTTP/0.6 Python/3.6.9\r\nContent-Length: ' + str(
                        len(
                            message)) + '\r\nUOSP: True\r\nUOSP_VERSION: 0.1\r\nUOSP_VALIDITY:' + cache_address[1] + '\r\n\r\n' + message

                    self.connection.send(data.encode())
                transaction.commit()
                db.close()
                if not self.cache_request:
                    # Can be treated identical to a regular socket object
                    if validators.url('http://' + http_header['host']):

                        remote.connect((http_header['host'], http_header['port']))
                        ssl_remote = ssl.wrap_socket(remote, ca_certs=None, cert_reqs=ssl.CERT_NONE)

                        # send request to the requested resource with UOSP header added
                        request_string = 'GET / HTTP/1.1\r\nHost: ' + http_header['host'] + ':' + str(
                            http_header['port']) + '\r\nUOSP: True\r\nUOSP_VERSION:' + UOSP_VERSION + '\r\n\r\n'
                        ssl_remote.sendall(request_string.encode())

                        # trigger and handle state change
                        self.handle_state_change(request_string)
                        self.exchange_loop(self.connection, ssl_remote)
                    else:
                        logging.error("The URL specified wasn't valid")
            else:
                # relay request to remote http server
                try:
                    logging.info('Non-onion traffic, relay HTTP traffic as a normal
                    ↪  proxy would do')
                    remote = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                    remote.connect((http_header['host'], http_header['port']))
                    logging.info('Connected to %s %s' % (http_header['host'],
                    ↪  http_header['port']))
                    request_string = 'GET / HTTP/1.1\r\nHost: ' + http_header['host']
                    ↪  + ':' + str(
                        http_header['port']) + '\r\n\r\n'
                    remote.sendall(request_string.encode())
                    self.legacy_request = True
                    self.exchange_loop(self.connection, remote)
                except Exception as e:
                    print(e)
        else:
            logging.error("Couldn't parse request header")

    """
    Loop for exchanging data between client and remote server

    :param client Handle for client socket
    :param remote Handle for server socket
    """

    def exchange_loop(self, client, remote):
        while True:
            # wait until client or remote is available for read
            r, w, e = select.select([client, remote], [], [])

            # there is data to be read from the client socket
            if client in r:
                # receive data and parse its body
                parsed_data = self.parse_http_body(client)
                if parsed_data is not None:
                    # trigger protocol state change2
                    self.handle_state_change(parsed_data)

            # there is data to be read from the server socket
            if remote in r:
                # check if legacy request
                if not self.legacy_request:
                    # receive data and parse its body
                    parsed_data = self.parse_http_body(remote)
                    if parsed_data is not None:
                        # check if UOSP response
                        if not parsed_data[1]:
                            # no UOSP response, just send back the data received
                            data = 'HTTP/1.1 200 OK\r\nServer: BaseHTTP/0.6
                            ↪  Python/3.6.9\r\nContent-Length: ' + str(
                                len(parsed_data[2])) + '\r\n\r\n' + parsed_data[2]
                            client.send(data.encode())
                            break
                        else:
                            # UOSP response, trigger protocol state change
                            self.handle_state_change(parsed_data, remote)
                    # don't send anything back to the client, till the UOSP has
                    ↪  finished, then reply with the temporary onion address
                    if self.csm.state == 'server_fin_received':
```

```python
                        # data = 'HTTP/1.1 301 Moved Permanently\r\nLocation: ' + 'http://' + str(self.uosp_address) + '.onion\r\n\r\n'
                        message = "<HTML><HEAD><TITLE>UOSP Response</TITLE></HEAD><BODY><h3>UOSP</h3>This page supports the " \
                                  "Unlinkable Onion Service Protocol (UOSP). Please use the following link <a href='http://" + str(
                            self.uosp_address) + ".onion'> USOP Link</a></BODY></HTML>"

                        data = 'HTTP/1.1 200 OK\r\nServer: BaseHTTP/0.6 Python/3.6.9\r\nContent-Length: ' + str(
                            len(
                                message)) + '\r\nUOSP: True\r\nUOSP_VERSION: 0.1\r\nUOSP_VALIDITY:' + self.uosp_validity + '\r\n\r\n' + message

                        client.send(data.encode())
                        # add uosp data address to db for the given intro address
                        db = MyZODB(DB_PATH)
                        if self.intro_host.split('.')[0] not in db.dbroot:
                            db.dbroot[self.intro_host.split('.')[0]] = UospIntroAddress(
                                self.intro_host.split('.')[0])
                        db.dbroot[self.intro_host.split('.')[0]].add_data_address(self.uosp_address, self.uosp_validity)
                        transaction.commit()
                        db.close()
                        break
                    # handle uosp data response
                    if self.csm.state == 'data_resp_received':
                        data = 'HTTP/1.1 200 OK\r\nServer: BaseHTTP/0.6 Python/3.6.9\r\nContent-Length: ' + str(
                            len(parsed_data[2])) + '\r\nUOSP:True\r\nUOSP_VERSION:0.1\r\n\r\n' + parsed_data[2]
                        client.send(data.encode())
                else:
                    # legacy request send back the response to the client
                    data = remote.recv(4096)
                    if client.send(data) <= 0:
                        break


if __name__ == '__main__':
    ForkingTCPServer.allow_reuse_address = True
    with ForkingTCPServer((PROXY_HOST, PROXY_PORT), SocksProxy) as server:
        server.serve_forever()
```

A.8 Proxy Tor Configuration

```
## Logging
Log notice file /var/log/tor/notices.log
Log info file /var/log/tor/info.log
Log debug file /var/log/tor/debug.log
## Run as daemon in background
RunAsDaemon 1
```

Appendix B: Performance Analysis

B.1 Analysis Module

```python
import time
import csv
import re
import datetime

# default timeout after 3 minutes (180s)
TIMEOUT = 180

# default number of intro points is currently 3
NUM_OF_INTRO_POINTS = 3

from connection.utils import establish_connection
from stem.control import EventType
from circuitAnalyzer import circCallback
from logAnalyzer import logCallback_v3
from logAnalyzer import logCallback_v2
from descriptorAnalyzer import descCallback
from logAnalyzer import thread_intro_circuit_q
from circuitAnalyzer import thread_circuit_list_q
from circuitAnalyzer import thread_upload_circuit_q
from descriptorAnalyzer import thread_descriptors_q
from circuitAnalyzer import circMinorCallback
from connection.utils import create_ephemeral_hidden_service
from connection.utils import create_hidden_service

# password for control port authentication
PASSWORD = 'tor'

# regex for ipv4 address in desc event
UPLOAD_REGEX = '([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}) ([0-9]{3,4})'

# V3
NUM_OF_CREATE_EVENT_V3 = 2
NUM_OF_UPLOAD_EVENTS_V3 = 16
NUM_OF_UPLOADED_EVENTS_V3 = 16

# V2
NUM_OF_CREATE_EVENT_V2 = 2
NUM_OF_UPLOAD_EVENTS_V2 = 6
NUM_OF_UPLOADED_EVENTS_V2 = 6

controller = None

# filepath of output csv file
CSV_PATH = "/tmp/provisioning_time_" + time.strftime("%Y_%m_%d_%H_%M",
↪   time.localtime()) + ".csv"


def checkDictEntries(dictionary, csv_field_name):
    for entry in csv_field_name:
        if entry not in dictionary:
            dictionary[entry] = "FAIL"


def start(version, ephemeral, timeout, num_intro_points):
    global controller, TIMEOUT, NUM_OF_INTRO_POINTS
    csv_dict = {}
    startTime = time.time()

    # placeholder for the lists
    intro_point_list = []
    descriptor_create_list = []
    descriptor_upload_list = []
    descriptor_uploaded_list = []
    hsdir_upload_circuit_list = []

    # if timeout is provided by parameters use this one, otherwise the default timeout (360s) is used
    if timeout is not None and isinstance(timeout, int):
        TIMEOUT = timeout

    # if a custom number of intro points is provided, use it, otherwise use the default (3)
    if num_intro_points is not None and isinstance(timeout, int):
        NUM_OF_INTRO_POINTS = num_intro_points

    # csv header names
    CSV_FIELD_NAME_V3 = ['service_initiated']

    for i in range(1, NUM_OF_INTRO_POINTS + 1):
        CSV_FIELD_NAME_V3.append('intro_' + str(i) + '_build_start')
        CSV_FIELD_NAME_V3.append('intro_' + str(i) + '_build_end')

    for i in range(1, NUM_OF_INTRO_POINTS + 1):
        CSV_FIELD_NAME_V3.append('intro_' + str(i) + '_established')

    CSV_FIELD_NAME_V3.extend(['desc_1_created', 'desc_2_created', 'upload_hs_1',
                              'uploaded_hs_1',
                              'upload_hs_2', 'uploaded_hs_2', 'upload_hs_3', 'uploaded_hs_3', 'upload_hs_4',
                              'uploaded_hs_4',
                              'upload_hs_5', 'uploaded_hs_5', 'upload_hs_6', 'uploaded_hs_6', 'upload_hs_7',
                              'uploaded_hs_7',
                              'upload_hs_8', 'uploaded_hs_8', 'upload_hs_9', 'uploaded_hs_9', 'upload_hs_10',
                              'uploaded_hs_10', 'upload_hs_11', 'uploaded_hs_11', 'upload_hs_12', 'uploaded_hs_12',
                              'upload_hs_13',
                              'uploaded_hs_13', 'upload_hs_14', 'uploaded_hs_14', 'upload_hs_15', 'uploaded_hs_15',
                              'upload_hs_16',
                              'uploaded_hs_16', 'upload_hs_1_hsdir_addr', 'upload_hs_2_hsdir_addr',
                              'upload_hs_3_hsdir_addr',
                              'upload_hs_4_hsdir_addr', 'upload_hs_5_hsdir_addr', 'upload_hs_6_hsdir_addr',
                              'upload_hs_7_hsdir_addr', 'upload_hs_8_hsdir_addr', 'upload_hs_9_hsdir_addr',
                              'upload_hs_10_hsdir_addr', 'upload_hs_11_hsdir_addr', 'upload_hs_12_hsdir_addr',
                              'upload_hs_13_hsdir_addr', 'upload_hs_14_hsdir_addr', 'upload_hs_15_hsdir_addr',
                              'upload_hs_16_hsdir_addr'])

    for i in range(1, 17):
        CSV_FIELD_NAME_V3.append(f"upload_hs_{i}_path")
        CSV_FIELD_NAME_V3.append(f"upload_hs_{i}_circuit_built")

    CSV_FIELD_NAME_V2 = ['service_initiated']

    for i in range(1, NUM_OF_INTRO_POINTS + 1):
        CSV_FIELD_NAME_V2.append('intro_' + str(i) + '_build_start')
        CSV_FIELD_NAME_V2.append('intro_' + str(i) + '_build_end')

    for i in range(1, NUM_OF_INTRO_POINTS + 1):
        CSV_FIELD_NAME_V2.append('intro_' + str(i) + '_established')

    CSV_FIELD_NAME_V2.extend(['desc_1_created', 'desc_2_created', 'upload_hs_1',
    ↪  'uploaded_hs_1',
                              'upload_hs_2', 'uploaded_hs_2', 'upload_hs_3',
                              ↪  'uploaded_hs_3', 'upload_hs_4',
                              'uploaded_hs_4',
                              'upload_hs_5', 'uploaded_hs_5', 'upload_hs_6',
                              ↪  'uploaded_hs_6', 'upload_hs_1_hsdir_addr',
                              'upload_hs_2_hsdir_addr', 'upload_hs_3_hsdir_addr',
                              'upload_hs_4_hsdir_addr', 'upload_hs_5_hsdir_addr',
                              ↪  'upload_hs_6_hsdir_addr'])

    for i in range(1, 7):
        CSV_FIELD_NAME_V2.append(f"upload_hs_{i}_path")
        CSV_FIELD_NAME_V3.append(f"upload_hs_{i}_circuit_built")

    # connect and authenticate
    controller = establish_connection(password=PASSWORD)
    csv_dict['service_initiated'] = datetime.datetime.now()

    # add event listeners
    controller.add_event_listener(circCallback, EventType.CIRC)
    controller.add_event_listener(descCallback, EventType.HS_DESC)
    controller.add_event_listener(circMinorCallback, EventType.CIRC_MINOR)
    if version == 3:
        controller.add_event_listener(logCallback_v3, EventType.DEBUG)
    else:
        controller.add_event_listener(logCallback_v2, EventType.DEBUG)

    # create hidden service
    if ephemeral == False and version == 3:
        create_hidden_service(controller, '/var/lib/tor', 80)
    else:
        create_ephemeral_hidden_service(controller, 80, 80, version)

    print(" * Onion Service created")

    # number of required events for V2/V3
    if version == 3:
        createEvent = NUM_OF_CREATE_EVENT_V3
        uploadEvents = NUM_OF_UPLOAD_EVENTS_V3
        uploadedEvents = NUM_OF_UPLOADED_EVENTS_V3
    else:
        createEvent = NUM_OF_CREATE_EVENT_V2
        uploadEvents = NUM_OF_UPLOAD_EVENTS_V2
        uploadedEvents = NUM_OF_UPLOADED_EVENTS_V2

    # loop while the required data is received
```

```python
    while True:
        if time.time() - startTime > TIMEOUT:
            print(" * One of many of the uploads timed out")
            break

        if thread_circuit_list_q.qsize() > 1:
            circuit_list = thread_circuit_list_q.get()
            thread_circuit_list_q.queue.clear()

        if thread_intro_circuit_q.qsize() > 1:
            intro_point_list = thread_intro_circuit_q.get()
            thread_intro_circuit_q.queue.clear()

        if thread_descriptors_q.qsize() > 3:
            descriptor_create_list = thread_descriptors_q.get()
            descriptor_upload_list = thread_descriptors_q.get()
            descriptor_uploaded_list = thread_descriptors_q.get()
            thread_descriptors_q.queue.clear()

        if not thread_upload_circuit_q.empty():
            hsdir_upload_circuit_list.append(thread_upload_circuit_q.get())

        if len(intro_point_list) >= 3 and len(descriptor_create_list) == 2 and len(
                descriptor_upload_list) == uploadEvents and len(descriptor_uploaded_list) == uploadedEvents:
            print(" * All required data available, starting correlation")
            break

    # clean up, remove event listeners (commented, because it was blocking some times, don't know why)
    controller.remove_event_listener(circCallback)
    controller.remove_event_listener(descCallback)
    controller.remove_event_listener(circMinorCallback)
    if version == 3:
        controller.remove_event_listener(logCallback_v3)
    else:
        controller.remove_event_listener(logCallback_v2)

    """
    correlates an element form the uploaded list with a provided fingerprint and returns the matched element

    :param uploaded_list: List of uploaded descriptors
    :param fingerprint: Fingerprint from the upload list to correlate
    :return: Element from the uploaded list that has the same fingerprint as the element from the upload list
    """

    def correlateFingerprint(uploaded_list, fingerprint):
        for elem in uploaded_list:
            if elem.fingerprint == fingerprint:
                return elem
        return None

    # read data from the various lists and put the into a dict uniformly that their values can be written to a CSV file
    for i in range(0, createEvent):
        csv_dict["desc_" + str(i + 1) + "_created"] = (descriptor_create_list[i]).getTime()
        i += 1

    for i in range(0, uploadEvents):
        if i <= len(descriptor_upload_list) - 1:
            csv_dict["upload_hs_" + str(i + 1)] = descriptor_upload_list[i]
            try:
                hsdir_addr = re.search(UPLOAD_REGEX, str(
                    controller.get_network_status
                    ↪   (relay=descriptor_upload_list[i].getFingerprint())))
                if hsdir_addr:
                    csv_dict["upload_hs_" + str(i + 1) + "_hsdir_addr"] =
                    ↪   str(hsdir_addr.group(1)) + ":" + str(
                        hsdir_addr.group(2))
                else:
                    csv_dict["upload_hs_" + str(i + 1) + "_hsdir_addr"] = "FAIL"
            except:
                csv_dict["upload_hs_" + str(i + 1) + "_hsdir_addr"] = "FAIL"
        else:
            csv_dict["upload_hs_" + str(i + 1)] = 'FAIL'
            csv_dict["upload_hs_" + str(i + 1) + "_hsdir_addr"] = 'FAIL'

    for i in range(0, len(hsdir_upload_circuit_paths)):
        # to upload circuit path is contained in the fingerprint attribute, while the
        ↪   upload circuit built timestamp is contained in the objects time stamp
        csv_dict[f"upload_hs_{i + 1}_circuit_built"] =
        ↪   hsdir_upload_circuit_list[i].getTime()
        csv_dict[f"upload_hs_{i + 1}_path"] =
        ↪   str(hsdir_upload_circuit_list[i].getFingerprint())

    for i in range(0, uploadedEvents):
        if i <= len(descriptor_uploaded_list) - 1:
            if csv_dict["upload_hs_" + str(i + 1)] != 'FAIL' and
            ↪   correlateFingerprint(descriptor_uploaded_list, str(
                    csv_dict["upload_hs_" + str(i + 1)].getFingerprint())):
                csv_dict["uploaded_hs_" + str(i + 1)] =
                ↪   correlateFingerprint(descriptor_uploaded_list, str(
                    csv_dict["upload_hs_" + str(i + 1)].getFingerprint())).getTime()
                csv_dict["upload_hs_" + str(i + 1)] = csv_dict["upload_hs_" + str(i +
                ↪   1)].getTime()
            else:
                csv_dict["upload_hs_" + str(i + 1)] = "FAIL"
        else:
            if csv_dict["upload_hs_" + str(i + 1)] != 'FAIL':
                csv_dict["upload_hs_" + str(i + 1)] = csv_dict["upload_hs_" + str(i +
                ↪   1)].getTime()
            csv_dict["uploaded_hs_" + str(i + 1)] = 'FAIL'

    for i in range(0, NUM_OF_INTRO_POINTS):
        if i <= len(intro_point_list) - 1:
            csv_dict["intro_" + str(i + 1) + "_established"] = intro_point_list[i][0]
            csv_dict["intro_" + str(i + 1) + "_build_start"] =
            ↪   circuit_list[str(intro_point_list[i][1])][0].getTime()
            csv_dict["intro_" + str(i + 1) + "_build_end"] =
            ↪   circuit_list[str(intro_point_list[i][1])][-1].getTime()
        else:
            csv_dict["intro_" + str(i + 1) + "_established"] = "FAIL"
            csv_dict["intro_" + str(i + 1) + "_build_start"] = "FAIL"
            csv_dict["intro_" + str(i + 1) + "_build_end"] = "FAIL"

            # write output to csv file
    print(" * Correlation performed, writing results to CSV file")

    # check if all mandatory fields are set, if not set them to FAIL
    if version == 3:
        checkDictEntries(csv_dict, CSV_FIELD_NAME_V3)
    else:
        checkDictEntries(csv_dict, CSV_FIELD_NAME_V2)

    if version == 3:
        with open(CSV_PATH, 'w', newline='') as csvfile:
            logwriter = csv.DictWriter(csvfile, delimiter=',', quotechar='"',
            ↪  quoting=csv.QUOTE_MINIMAL,
                                           fieldnames=CSV_FIELD_NAME_V3)
            logwriter.writeheader()
            logwriter.writerow(csv_dict)
    else:
        with open(CSV_PATH, 'w', newline='') as csvfile:
            logwriter = csv.DictWriter(csvfile, delimiter=',', quotechar='"',
            ↪  quoting=csv.QUOTE_MINIMAL,
                                           fieldnames=CSV_FIELD_NAME_V2)
            logwriter.writeheader()
            logwriter.writerow(csv_dict)
```

B.2 Circuit Analyzer

```
import datetime
import re
import queue

# global dictionary for circuits
circuits = {}

# regex for fingerprint
FINGERPRINT_REGEX = '[A-z0-9]{40}'

# synchronized queue for inter thread communication
thread_circuit_list_q = queue.Queue()

# synchronized queue for tor upload circuits
thread_upload_circuit_q = queue.Queue()

"""
class for circuit elements
"""

class circuitElement:
    def __init__(self, time_added, fingerprint):
        self.__time_added = time_added
        self.__fingerprint = fingerprint

    def getTime(self):
        return self.__time_added

    def getFingerprint(self):
        return self.__fingerprint


"""
callback handler when a CIRC event takes place
```

```python
"""


def circCallback(event):
    global circuits
    # save timestamp of the event
    timestamp = datetime.datetime.now()

    # write to queue such that the main thread has access to the circuit list
    if len(circuits) > 0:
        # put new version into queue
        thread_circuit_list_q.put(circuits)

    # circuits for finding introduction points
    if event.purpose == "HS_SERVICE_INTRO":
        # Circuit is new, add new entry to global circuit dictionary
        if event.status == "LAUNCHED":
            circuits[event.id] = []
        # circuit is extended
        elif event.status == "EXTENDED":
            circuits[event.id].append(circuitElement(timestamp,
            ↪  extractFingerprint(event.path)))
        # circuit failed or closed, delete entry from dictionary
        elif event.status == "FAILED" or event.status == "CLOSED":
            circuits.pop(event.id)

    if event.purpose == "HS_SERVICE_HSDIR":
        if event.status == "LAUNCHED":
            print(f"[{datetime.datetime.now()}] CircuitEvent: {event.id} | status:
            ↪  {event.status} | purpose: {event.purpose} | path: {event.path}")
        if event.status == "BUILT":
            # in this case the fingeprint attribute is misued for the upload circuit
            ↪  path
            thread_upload_circuit_q.put(circuitElement(timestamp, event.path))
            print(f"[{datetime.datetime.now()}] CircuitEvent: {event.id} | status:
            ↪  {event.status} | purpose: {event.purpose} | path: {event.path}")
        elif event.status == "FAILED":
            # One Upload failed, so a new attempt will be made.
            # thread_upload_circuit_q.put("FAIL")
            print(f"[{datetime.datetime.now()}] CircuitEvent: {event.id} | status:
            ↪  {event.status} | purpose: {event.purpose} | path: {event.path}")


"""
callback handler when a CIRC MINOR event takes place
"""


def circMinorCallback(event):
    global circuits
    if not event.id in circuits:
        if event.event == 'CANNIBALIZED':
            path_list = extractFingerprint(event.path, cannibalized=True)
            circuits[event.id] = []
            for elem in path_list:
                circuits[event.id].append(circuitElement(str(datetime.datetime.now())
                ↪  + " (CAN)", elem))


"""
```

```python
extracts the fingerprint from a path string that is not yet in the specific circuits
↪  lists

:param path: Path string of the CIRC event
:return: Fingerprint
"""


def extractFingerprint(path, cannibalized=False):
    fingerprint_re = re.compile(FINGERPRINT_REGEX)
    path_list = re.findall(fingerprint_re, str(path))

    if len(path_list) > 0 and not cannibalized:
        return path_list[-1]
    elif len(path_list) > 0 and cannibalized:
        return path_list


"""
print all the circuits and their paths
"""


def printCircuitList():
    for circuit in circuits:
        print(f"# {circuit}")
        for relay in circuits[circuit]:
            print(f" {relay.getTime()}: [{relay.getFingerprint()}]")
        print("\n")
```

B.3 Log Analyzer

```python
"""
callback handler when a LOG event takes place
"""

import datetime
import queue
import re

# regex for intro points
INTRO_ESTABLISHED_REGEX_V3 = 'service_handle_intro_established\(\): Successfully
↪  received an INTRO_ESTABLISHED cell on circuit ([0-9]{1,}) \(id: ([0-9]{1,})\)'
INTRO_ESTABLISHED_REGEX_V2 = 'rend_service_intro_established\(\): Received
↪  INTRO_ESTABLISHED cell on circuit ([0-9]{1,}) \(id: ([0-9]{1,})\)'

intro_circuit_list = []

# synchronized queue for inter thread communication
thread_intro_circuit_q = queue.Queue()

"""
callback handler when a LOG event takes place
"""

def logCallback_v2(event):
    global intropoints_extracted

    timestamp = datetime.datetime.now()
    # extractIntroEstablishedCircuit(event.message)
    int_tmp = extractCircuitID(event.message, timestamp, 2)
    if int_tmp is not None:
        intro_circuit_list.append(int_tmp)
        # put new version into queue
        thread_intro_circuit_q.put(intro_circuit_list)


def logCallback_v3(event):
    global intropoints_extracted

    timestamp = datetime.datetime.now()
    # extractIntroEstablishedCircuit(event.message)
    int_tmp = extractCircuitID(event.message, timestamp, 3)
    if int_tmp is not None:
        intro_circuit_list.append(int_tmp)
        # put new version into queue
        thread_intro_circuit_q.put(intro_circuit_list)


"""
extract the circuit ID from a intro established log message and return a list with
↪  its timestamp and id
:param line: Log string
:param timestamp: timestamp when the controller received the log message
:return: list of timestanp and ID of intro circuit (global ID)
"""

def extractCircuitID(line, timestamp, version):
    if version == 3:
        circuit_id = re.search(INTRO_ESTABLISHED_REGEX_V3, str(line))
    else:
        circuit_id = re.search(INTRO_ESTABLISHED_REGEX_V2, str(line))

    # if there is a match, extract the circuit ID
    if circuit_id is not None:
        return [timestamp, circuit_id.group(2)]
    else:
        return None
```

B.4 Descriptor Analyzer

```
import queue
import datetime

# synchronized queue for inter thread communication
thread_descriptors_q = queue.Queue()

descUploadEvents = []
descUploadedEvents = []
descCreatedEvents = []

"""
class for descriptor elements
"""


```

```python
class descElement:
    def __init__(self, time_added, fingerprint):
        self.time_added = time_added
        self.fingerprint = fingerprint

    def getTime(self):
        return self.time_added

    def getFingerprint(self):
        return self.fingerprint


"""
callback handler when a DESC event takes place
"""


def descCallback(event):
    global descEvents
    # collected all required data, put it in a queue such that the main thread can
    ↪ read them
    # put new version into queue
    thread_descriptors_q.put(descCreatedEvents)
    thread_descriptors_q.put(descUploadEvents)
    thread_descriptors_q.put(descUploadedEvents)

    if event.action == "CREATED":
        descCreatedEvents.append(descElement(datetime.datetime.now(), None))
    elif event.action == "UPLOAD":
        descUploadEvents.append(descElement(datetime.datetime.now(),
        ↪ event.directory_fingerprint))
        print(f"[{datetime.datetime.now()}] action: {event.action} | descriptor:
        ↪ {event.descriptor_id} | replica: {event.replica}")
    elif event.action == "UPLOADED":
        descUploadedEvents.append(descElement(datetime.datetime.now(),
        ↪ event.directory_fingerprint))
        print(f"[{datetime.datetime.now()}] action: {event.action} | descriptor:
        ↪ {event.descriptor_id} | replica: {event.replica}")
```

B.5 Start Analysis

```python
import re
from analysis import start
from connection.utils import establish_connection
import sys

NOTICES_LOG_PATH = '/var/log/tor/notices.log'
BOOTSTRAP_REGEX = 'Bootstrapped 100\%'
PASSWORD = 'tor'

version = int(sys.argv[1])
ephemeral = sys.argv[2] == "true" or sys.argv[2] == "True" or sys.argv[2] == "TRUE"
timeout = int(sys.argv[3])
num_intro_points = int(sys.argv[4])

bootstrapped = False


```

```python
def parseLog(file_path):
    global bootstrapped
    with open(file_path) as infile:
        for line in infile:
            if checkIfBoostrapped(line):
                bootstrapped = True

def checkIfBoostrapped(line):
    if re.search(BOOTSTRAP_REGEX, line) is not None:
        return True
    return False

while not bootstrapped:
    parseLog(NOTICES_LOG_PATH)

controller = establish_connection(password=PASSWORD)
print(f" * Number of circuits after bootstrapping: {len(controller.get_circuits())}")
print(" * Tor completely bootstrapped, starting analysis")

start(version, ephemeral, timeout, num_intro_points)
```

B.6 Utils

```python
import sys
import stem.control
import stem.connection

"""
Establish a connection to the control port of tor and authenticate
"""
CONTROL_PORT = 9051
CONTROL_ADDRESS = '127.0.0.1'


def establish_connection(address=CONTROL_ADDRESS, port=CONTROL_PORT, password=None):
    # Get connection to tor via the control port
    try:
        # the explicit declaration of the port isn't necessary since it's the default
        ↪  port
        controller = stem.control.Controller.from_port(address, port)
    except stem.SocketError as exc:
        print(f" * Unable to connect to port control port 9051 ({exc})")
        sys.exit(1)

    if controller:
        # authenticate
        try:
            if (password != None):
                controller.authenticate(password=password)
            else:
                controller.authenticate()
            print(" * Connection established and authenticated")
            return controller
        except stem.connection.PasswordAuthFailed:
            print(" * Unable to authenticate, password is incorrect")
            sys.exit(1)
        except stem.connection.AuthenticationFailure as exc:
            print(f" * Unable to authenticate: {exc}")
            sys.exit(1)


"""
Create new ephemeral Hidden Service and return its onion id
"""


def create_ephemeral_hidden_service(controller, source_port, target_port, version):
    try:
        if version == 2:
            hs = controller.create_ephemeral_hidden_service({source_port:
             ↪  target_port}, await_publication=False,
                                                            key_type="NEW",
                                                            ↪  key_content="RSA1024",
                                                            ↪  detached=False)
        else:
            hs = controller.create_ephemeral_hidden_service({source_port:
             ↪  target_port}, await_publication=False,
                                                            detached=False)
        if (hs.service_id):
            return hs.service_id
        else:
            return -1

    except stem.ControllerError as exc:
        print(f"Unable to create the hidden service ({exc})")
        sys.exit(1)


def create_hidden_service(controller, path, port):
    try:
        controller.create_hidden_service(path, port, target_address='127.0.0.1')
    except stem.ControllerError as exc:
        print(f"Unable to create the hidden service ({exc})")
        sys.exit(1)
```

B.7 Docker File

```
# image based on debian
FROM debian:latest

# update package lists
RUN apt-get update -y

# workaround for avoiding debconf failure messages
RUN echo 'debconf debconf/frontend select Noninteractive' | debconf-set-selections

# install all packages required to compile and run tor within the container
RUN apt-get install libssl-dev -y -qq > /dev/null
RUN apt-get install zlib1g-dev -y -qq > /dev/null
RUN apt-get install python3 -y -qq > /dev/null
RUN apt-get install python3-pip -y -qq > /dev/null
RUN apt-get install libevent-dev -y -qq > /dev/null

# create directories
RUN mkdir -p /home/script
RUN mkdir -p /home/tor
RUN mkdir -p /etc/tor
RUN mkdir -p /var/log/tor
RUN mkdir -p /var/lib/tor
RUN mkdir -p /run/tor

# init file, tor binaries and torrc file

# the init handles execution parameters (which analysis should be started, V2, V3,
↪  V3VN, V3NE?)
ADD init_tor.sh /home/script/init_tor.sh

# tor source code, which is compiled within the container (should be kept up-to-date)
ADD tor /home/tor

# tor config file
ADD torrc /etc/tor/torrc

# vanguards config file
ADD vanguards.conf /etc/tor/vanguards.conf

# python requirements (currently only stem)
ADD requirements.txt /home/requirements.txt

# install python package manager
RUN pip3 install --upgrade pip

# install required packages (currently only stem)
RUN pip3 install -r /home/requirements.txt

# install vanguards (could also be done within the requirements file)
RUN pip3 install vanguards

# create log files
RUN touch /var/log/tor/debug.log
RUN touch /var/log/tor/info.log
RUN touch /var/log/tor/notices.log

# build custom tor binary
RUN sh ./home/tor/configure
RUN chmod +x /home/tor/install-sh
RUN make && make install

RUN chmod +x /home/script/init_tor.sh
RUN chmod 700 /var/lib/tor

# add timing analysis prototype
ADD prototype-timing-analysis/ /home/timing-analysis/

# run init script
ENTRYPOINT ["/bin/bash","/home/script/init_tor.sh"]
```

B.8 Analysis Server

```
#!/bin/bash

target="/path/to/results"
image="tor-docker:latest"
timeout=180
```

```
introduction_points=3
for i in {1..N}
do
    echo "Iteration $i: V2"
    docker run -it --rm -v${target}V2:/results -v/etc/localtime:/etc/localtime:ro
    ↪  "${image}" -v 2 -t ${timeout} -i ${introduction_points}
    echo "Iteration $i: V3"
    docker run -it --rm -v${target}V3:/results -v/etc/localtime:/etc/localtime:ro
    ↪  "${image}" -v 3 -t ${timeout} -i ${introduction_points}
    echo "Iteration $i: V3NE"
    docker run -it --rm -v${target}V3NE:/results -v/etc/localtime:/etc/localtime:ro
    ↪  "${image}" -v 3NE -t ${timeout} -i ${introduction_points}
    echo "Iteration $i: V3VN"
    docker run -it --rm -v${target}V3VN:/results -v/etc/localtime:/etc/localtime:ro
    ↪  "${image}" -v 3VN -t ${timeout} -i ${introduction_points}
done
```

B.9 Tor Configuration

```
## Logging
## Send all messages of level 'notice' or higher to /var/log/tor/notices.log
Log notice file /var/log/tor/notices.log
## Send all messages of level 'info' possible message to /var/log/tor/info.log
Log info file /var/log/tor/info.log
## Send every possible message to /var/log/tor/debug.log
Log debug file /var/log/tor/debug.log

## Control port
ControlPort 9051
HashedControlPassword 16:CCA169830431CAB5608D58317D89A87D314B84B42FCCF3EDC82FD8B4E4

## Run as daemon in background
RunAsDaemon 1

## The directory for keeping all the keys/etc. By default, we store
## things in $HOME/.tor on Unix, and in Application Data\tor on Windows.
DataDirectory /var/lib/tor


SafeLogging 0
```

B.10 Initialization Script

```
##!/bin/bash

# get parameter
while getopts ":v:t:i:" opt; do
  case $opt in
    v) version="$OPTARG"
    ;;
    t) timeout="$OPTARG"
    ;;
    i) num_intro="$OPTARG"
    ;;
    \?) echo "Invalid option -$OPTARG" >&2
    ;;
  esac
```

```
done

if [[ -z ${version} ] || [-z ${timeout}] || -z ${num_intro};
then
        echo -e "\t\nUsage:\nbash init_tor.sh -v VERSION -t TIMEOUT -i
        ↪  NUM_INTRO_POINTS\n"
        echo -e "\t3: Ephemeral Onion Service Version 3\n"
        echo -e "\t3NE: Non-ephemeral Onion Service Version 3\n"
        echo -e "\t3VN: Ephemeral Onion Service with Vanguards Version 3\n"
        echo -e "\t2: Ephermeral Onion Service Version 2\n"
        exit 1
else
   # start tor
   tor -f /etc/tor/torrc --quiet &

   export PYTHONPATH="/home/timing-analysis"
   case "$version" in
        3)      python3 /home/timing-analysis/analysis/startAnalysis.py 3 "True"
        ↪  ${timeout} ${num_intro}
        ;;
        3NE)    python3 /home/timing-analysis/analysis/startAnalysis.py 3 "False"
        ↪  ${timeout} ${num_intro}
        ;;
        3VN)    vanguards --config /etc/tor/vanguards.conf &
                python3 /home/timing-analysis/analysis/startAnalysis.py 3 "True"
                ↪  ${timeout} ${num_intro}
        ;;
        2)      python3 /home/timing-analysis/analysis/startAnalysis.py 2 "True"
        ↪  ${timeout} ${num_intro}
        ;;
        *)      echo -e "Version can only be [2]|[3]|[3NE]|[3VN]"
                exit 1
        ;;
        esac
   # copy result file
   cp /tmp/provisioning_time* /results
fi
```