JOHANNES KEPLER
UNIVERSITY LINZ

Author
DI **Tobias Höller**, BSc
0957305

Submission
Institute of
Networks and Security

First Supervisor
Univ.Prof. Dr.
René Mayrhofer

Second Supervisor
Alastair Beresford, PhD

Assistant Thesis
Supervisor
Dr. **Michael Roland**, MSc

October 2022

A Privacy Preserving Networking Approach for Distributed Digital Identity Systems

Doctoral Thesis

to confer the academic degree of

Doktor der technischen Wissenschaften

in the Doctoral Program

Technische Wissenschaften

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

Distributed systems are widely considered more privacy friendly than centralized systems because there is no central authority with access to all of the information. However, this does not consider the importance of network privacy. If users establish peer-to-peer connections to each other, adversaries monitoring the network can easily find out who is communicating with whom, at which times, and for how long, even if the communication is end-to-end encrypted. For digital identity systems this is especially critical, because knowledge about when and where an individual uses their digital identity is equivalent with knowing what the individual is doing.

The research presented in this thesis strives to design a distributed digital identity system that remains resilient against passive adversaries by instrumenting the anonymity network Tor. Significant efforts were dedicated to analyze how suited the Tor network is for supporting such distributed systems by measuring the usage of onion services and the time needed to start a new onion service. While this analysis did not detect any privacy issues within the current Tor network, it revealed several shortcomings in regard to the network latency of Tor onion services, which are addressed in the final parts of this thesis. Several modifications are proposed that are shown to significantly reduce the waiting times experienced by users of privacy preserving distributed digital identity systems.

Kurzfassung

Allgemein wird oft davon ausgegangen, dass verteilte Systeme besser für die Privatsphäre von Benutzern sind als zentralisierte Anwendungen. Es gibt allerdings einen Aspekt der dabei nicht ausreichend berücksichtigt wird: Network Privacy. Bei der Verwendung von dezentralen Systemen müssen Benutzer Nachrichten direkt an andere Benutzer übermitteln, was es Angreifern, die Netzwerkkommunikation überwachen können, ermöglicht herauszufinden wer wann mit wem wie lange kommuniziert. Selbst der Einsatz von Ende-zu-Ende Verschlüsselung stellt hier für Angreifer kein Hindernis dar. Für manche Anwendungen, wie zum Beispiel digitale Identitäten, ist das besonders problematisch, weil die Information darüber wann und wo eine digitale Identität verwendet wurde das Bewegungs- und Aktivitätsprofil der betroffenen Person enthüllt.

Die Forschung, die in dieser Arbeit präsentiert wird, hat es sich zum Ziel gesetzt, ein verteiltes System für digitale Identitäten zu entwicklen, das derartigen Angriffen widerstehen kann. Zu diesem Zweck wird das Anonymisierungsnetzwerk Tor verwendet, dessen Analyse ein wesentlicher Teil dieser Arbeit ist. Durch die Analyse von Tor Onion Services, insbesondere wie sie verwendet werden und wie lange es dauert einen neuen Onion Service zu starten, konnte einerseits bestätigt werden, dass das Tor Netzwerk ein ausreichendes Level an Anonymität bietet, andererseits konnten mehrere Punkte identifiziert werden, in denen Tor Kommunikation länger als notwendig verzögert. Diese Erkenntisse führten zu mehreren Änderungsvorschlägen an der aktuellen Tor Implementierung, welche die von Benutzern erlebten Wartezeiten bei der Nutzung eines Privatesphäre bewahrenden verteilten Systems für digitalen Identitäten wesentlich reduzieren können.

Contents

Chapter 1

Introduction

1.1 Motivation

Digital services like online shopping or social networks have become an essential part of society over the last two decades. To support these use cases, users have grown more and more accustomed to using digital identities (email addresses, user accounts) for their day-to-day activities. Realizing the importance of these digital identities, nation states have started to provide their citizens with officially issued digital identities that are directly linked to one specific individual. The European eID [40] or the Indian Aadhaar [53] are two examples for such efforts that also highlight different approaches in regards to privacy.

The eID only supports purely digital authentication, meaning it can be used by citizens while using a digital device of their own to prove their identity. It does not replace traditional physical identity documents, because there exists no obvious link between a physically present individual and his/her digital identity. Aadhaar solves this problem by including the biometric information about every citizen in its database. This removes the need for physical identity documents because every individual can easily be identified by searching the Aadhaar database for matching biometric features. While the European Union decided against this approach—most likely because of privacy concerns—there are alternative ideas towards replacing physical identity documents with digital ones, the most prominent being the recent standard for mobile driving licenses [64].

For entities issuing identities, using digital identities has several benefits. They are cheaper because there is no need to print and distribute physical documents, they can be issued much faster, they are much harder to forge (as long as their digital signatures are properly verified) and verification can be automated. This last point is especially useful because it reduces the cost needed for officers that would otherwise have to verify physical identity documents. At the same time, users can also benefit in several ways: New documents can be issued instantly and outdated documents can be updated automatically, so problems like expired passports can be avoided. The automated verification process should most likely reduce waiting times at various checkpoints. Finally, digital identities enable a process of partial disclosure that is not really possible for physical documents. If an individual tries to purchase alcohol, for example, they are very likely asked to prove that their age is beyond a certain threshold. Presenting an officially issued physical identity document, like a driving license or a passport, fulfills that request, but it reveals much more information than necessary (e.g. place of birth, nationality, full name, ...). With digital documents it is possible to use selective disclosure to only reveal as much information as needed to the verifying entity.

Unfortunately, digital identities systems also bring several new issues. Both, the European eID and the Indian Aadhaar system require users to interact with

them, whenever they want to use their digital identity. This means that authorities suddenly get much more information about their citizens daily life. As a consequence, citizens need to trust their government to not abuse this information itself and to be technically capable enough to prevent this data from being abused by others. In the case of Aadhaar, there have been numerous reports in the past that indicate that the system does not protect the data of the Indian citizens as much as it should [42, 108, 116]. Additionally, Aadhaar decided to assign a global identifier to all identities that is used in every interaction [1], meaning that every Aadhaar transaction can also be linked to one specific Aadhaar identity by the entities asking for verification, further compromising the privacy of its users.

For endangered or privacy sensitive users with little trust in their government to protect their personal data, such solutions are unacceptable. The easiest way to convince them would be to implement a distributed digital identity system which stores information on user controlled devices and transactions do not involve the authority that originally issued the identity. One example of this approach is the current mDL standard that stores all identity information locally on the user's device and can verify the user's identity offline. However, mDL can also verify identities online [63] and as soon as that happens, a new challenge emerges:

Distributed systems require user controlled devices to establish connections to every service that asks for the user's identity. Entities with access to a large share of network traffic like Internet service providers can observe who is communicating with whom and authorities can take legal action to obtain this privacy sensitive information from them. At this point, distributed digital identity systems face the same disadvantage as the centralized ones. Authorities can learn when and where individuals use their identities. Past revelations by Edward Snowden [88] prove that authorities are capable and willing to force such entities to provide them with access to all the data they are interested in. And more recent efforts of the European Union show that they are also willing to pass legislation that actively weakens the security and privacy of all users [38].

The goal of this research is to find a networking approach that allows the creation of a distributed digital identity system that protects the privacy of all users by preventing malicious actors, even if they have access to significant shares of network traffic, from learning anything about how an individual is using their digital identity.

1.2 Objectives

The objective of our research is to find a networking scheme for an identity system that meets the following requirements:

1. Distributed,
2. privacy-preserving,
3. scalable,
4. low-latency,
5. and feasible.

Distributed means that we want the networking scheme to not rely on any central devices or entities that are involved in most ongoing transactions. This is not simply satisfied by having devices connect to each other directly, support

infrastructure like DNS or certification authorities must also be taken into account. One specific challenge originating from this requirement is the question of how devices can learn about each other, without introducing a centralized entity.

Preserving user privacy means that entities observing network traffic should not be able to learn anything significant about how individuals are using their digital identities. Additionally, the privacy of users should also not be negatively impacted if attackers are able to actively modify traffic by either injecting, modifying or dropping network packets. Beyond that, attackers might also operate public services like DNS or NTP to obtain additional data. If such information can be correlated with observed network traffic to compromise the privacy of users, the network would also not be privacy preserving.

In order to support digital identity systems that provide everyone on Earth with an individual identity, the networking approach has to scale well enough to support several billion users. At the same time, the devices of all those users will have to communicate with other devices/services that need information about the users. Therefore, the total number of devices active in such a network can be expected to be much higher than the number of enrolled users. To support such a large distributed system, the complexity of a single transaction must be independent of the size of the entire distributed system.

At the same time, we have to acknowledge that the user experience of a distributed digital identity system would be very negatively impacted by high latency. Procedures that require users to prove their identity can usually only be completed, after the identity has been successfully verified. This forces users to wait, and the longer the waiting periods are, the less likely users are to accept the system. The success of contactless payment systems has shown that transaction times between 300 and 500 ms [3] are not perceived as annoying by the general public. While we believe it to be unreasonable to expect a distributed system to match those times, user acceptance depends on transaction times being as close as possible to those of alternative technologies. To also provide an upper limit, the EU found that transaction times for border checks would be acceptable as long as they remain below 30 seconds [39]. However, network latency is just one of the contributing factors of transaction time, so the network latency will have to be significantly lower than 30 seconds to leave room for all the other tasks of a digital identity system.

Finally, the desired networking scheme must be feasible to implement at this point in time. It must not depend on technologies that do not exist yet or technologies that exist in theory without ever having been deployed in practice. This ensures that we can deploy and test the proposed networking approach in practice, which in turn enables us to collect data to quantify how close the proposed solution is to fulfilling the research objectives.

1.2.1 Non-Objectives

The scope of this thesis is strictly limited to the networking approach for a privacy preserving distributed digital identity system. Questions like how biometric comparison can be done while remaining private, or how digital signatures can be created without revealing the identity of the signer are also crucial for the design of a privacy preserving digital identity system, but they are not in scope of this research.

1.3 Approach

We relied on the following methods to achieve the results presented in this thesis:

- literature review,
- evaluation of protocols and their implementations,
- design and implementation of prototypes,
- design and execution of experiments,
- and data analysis.

1.4 Contributions

The contributions presented in this thesis can logically be split into two different categories: First, are the theoretic contributions in what the network protocols for a privacy preserving distributed system should look like and all the potential pitfalls that might cause unintended privacy leaks. The most prominent contributions in this area are approaches for private service discovery to allow devices to learn about each other without leaking privacy sensitive information and the evaluation of network anonymization tools that would be suitable to support such systems.

Second, are several contributions surrounding Tor, the anonymization network that we identified to be the best currently available system to provide network anonymity. This includes evaluations of the Tor network in terms of performance, privacy, and reliability. The investigation into Tor onion services was acknowledged as valuable by the Tor project itself and at their request, a summary was published directly on the official blog of the Tor project [58]. Our results provided new insights into how onion services are being used in practice.

Research into the reliability of the Tor network also provided valuable results. While conducting this research, a group of suspiciously acting relays could be identified. Further investigation led to the conclusion that this group of relays was most likely secretly operated by a single entity violating the rules for Tor relay operators. Those findings were officially disclosed to the Tor project and they were able to confirm that these relays were in fact acting maliciously [99] and removed them from the network [74], so this research also improved the security of the Tor network as a whole. This discovery of malicious Tor relays was also reported in the media [4, 10, 21, 71, 77, 103].

During the final stages of this research, the lessons learned from the extensive analysis of the Tor network were used to suggest and implement several specific changes that stand to improve its capability to support privacy preserving distributed systems. Those contributions are not limited to digital identity systems, they are also capable of supporting various other scenarios that require a distributed system that ensures network privacy.

1.5 Publications

Parts of this thesis have already undergone peer review and were published in scientific workshops, conferences, and journals:

- T. Höller: Towards establishing the link between a person's real-world interactions and their decentralized, self-managed digital identity in the Digidow architecture, in IDIMT-2019: Innovation and Transformation in a Digital World, Kutná Hora, Czech Republic, Trauner Verlag, pp. 327–332, 2019. ISBN 978-3-99062-590-3.
- R. Mayrhofer, M. Roland, and T. Höller: Poster: Towards an Architecture for Private Digital Authentication in the Physical World, in Network and Distributed System Security Symposium (NDSS Symposium 2020), Posters, San Diego, CA, USA, 2020.
- T. Höller, T. Raab, M. Roland, and R. Mayrhofer: On the feasibility of short-lived dynamic onion services, in 2021 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, IEEE, pp. 25–30, 2021.
- T. Höller, M. Roland, and R. Mayrhofer: On the state of V3 onion services, in Proceedings of the ACM SIGCOMM 2021 Workshop on Free and Open Communications on the Internet (FOCI '21), Virtual, ACM, pp. 50–56, 2021.
- T. Höller, M. Roland, and R. Mayrhofer: Analyzing inconsistencies in the Tor consensus, in The 23rd International Conference on Information Integration and Web Intelligence (iiWAS2021), Linz, Austria, ACM, pp. 487–496, 2021.
- T. Höller, M. Roland, and R. Mayrhofer: Evaluating Dynamic Tor Onion Services for Privacy Preserving Distributed Digital Identity Systems, Journal of Cyber Security and Mobility 11, 2, pp. 141–164, 2022. ISSN 2245-1439.

One additional publication (based on the work presented in chapter 7) is currently undergoing peer-review and expected to be published in the future:

- T. Höller, M. Roland, and R. Mayrhofer: Optimizing Tor onion services for Tor-aware pub/sub based communication. In Proceedings on Privacy Enhancing Technologies Symposium 2023.

Apart from scientific publications, the research presented in this thesis has also produced a significant amount of source code that analyzes and visualizes collected data. As it is infeasible to include all the data and source code within a printed thesis, this supporting material is published on GitHub instead. The repository https://github.com/TTH-Someone-stole-my-name/privacy-preserving-networking-for-distributed-systems.git contains all the tools developed while working on this thesis and all the data that can be included without compromising the privacy of third parties[1].

1.6 Outline

This thesis is structured to present the conducted research as a series of consecutive steps that continue to build on top of each other. In chapter 2 we provide the necessary background information on the Digidow project, the research endeavor that motivated this research. We also include a discussion of network anonymity, different strategies on how to achieve it, and available implementations. Finally, we describe the functionality of the Tor network in greater detail.

The main contributions of this thesis are presented in the chapters 3 to 7. Chapter 3 elaborates the proposed network architecture of privacy preserving distributed digital identity systems. It includes discussions of network interactions, potential adversaries, and the service discovery process responsible for

[1] Since one publication is still under review, material regarding the work presented in section 7 will be added later

detecting other devices within a distributed setup. Based on this information, chapter 4 presents the work conducted to evaluate the degree of privacy already offered by the Tor network and highlights privacy leaks that still exist in the current design. To mitigate some of the issues identified in the two previous chapters, chapter 5 investigates the concept of dynamic onion services.

Chapter 6 contains a detailed analysis of the Tor network, to determine if its structure and participants can be considered trustworthy enough to meet the specified objectives. This part of the thesis was originally a side track that was triggered while conducting experiments on the live Tor network for chapter 4 when parts of the Tor network were found to not behave according to the specification.

Finally, chapter 7 combines all the information gathered from the previous chapters to propose several changes to the current Tor implementation that can improve the latency and privacy of distributed systems built on top of the Tor network. Chapter 8 wraps up the thesis by summarizing the results and describing future work that could be done to further advance scientific knowledge in this field but was beyond the scope of this thesis.

Chapter 2

Background

2.1 The Digidow Project

The Digidow project[1] is a research project funded by the Christian Doppler Gesellschaft[2], the Austrian Ministry for Digitalization[3], 3 Banken IT[4], ekey[5], Kepler Universitätsklinikum[6], NXP Semiconductors Austria[7], and the Austrian State Printing House[8]. The entire Digidow projects operates on three assumptions about the future that are essential to understand its goals:

1. The use of digital identities will continue to increase over time.
2. Biometric identification will become the primary method of linking physical individuals to digital identities in the future.
3. Users will demand access to their digital identities, even when they are not carrying any physical identity documents or devices with them.

Considering that there are numerous systems that implement these assumptions for ticketing in public transport [104, 105, 106], these assumptions must be considered reasonable. Provided with those expectations, one already established digital identity system immediately comes to mind: The Indian Aadhaar [53] system already supports the requirements listed above, leaving no doubts about their feasibility. The question we should ask instead, is how to best protect the privacy of citizens if such a digital identity system is implemented. As the research for this thesis was conducted mostly within the Digidow project, most of the objectives listed in section 1.2 can be mapped to explicit goals of the Digidow project.

The need for a distributed networking scheme originates from Digidow's assumption that a central database with biometric information about millions of citizens is unacceptable because it puts too much power and responsibility on the operators. Beyond that, an essential part of privacy is a user's agency over their own data. If data is centrally stored and managed by a single entity, users lose control of their own information.

Figure 2.1 visualizes the envisioned network architecture expected by the Digidow project. The three most important actors in Digidow (and every other digital identity system relying on biometric features) are *personal identity agents (PIAs)*, biometric *sensors*, and *verifiers*, as they are needed every time a digital identity is used. Personal identity agents are a piece of software the holds all the

[1] https://www.digidow.eu
[2] https://www.cdg.ac.at/
[3] https://www.bmdw.gv.at/
[4] https://www.3bankenit.at/
[5] https://www.ekey.net/
[6] https://www.kepleruniklinikum.at/
[7] https://www.nxp.com/
[8] https://www.staatsdruckerei.at/

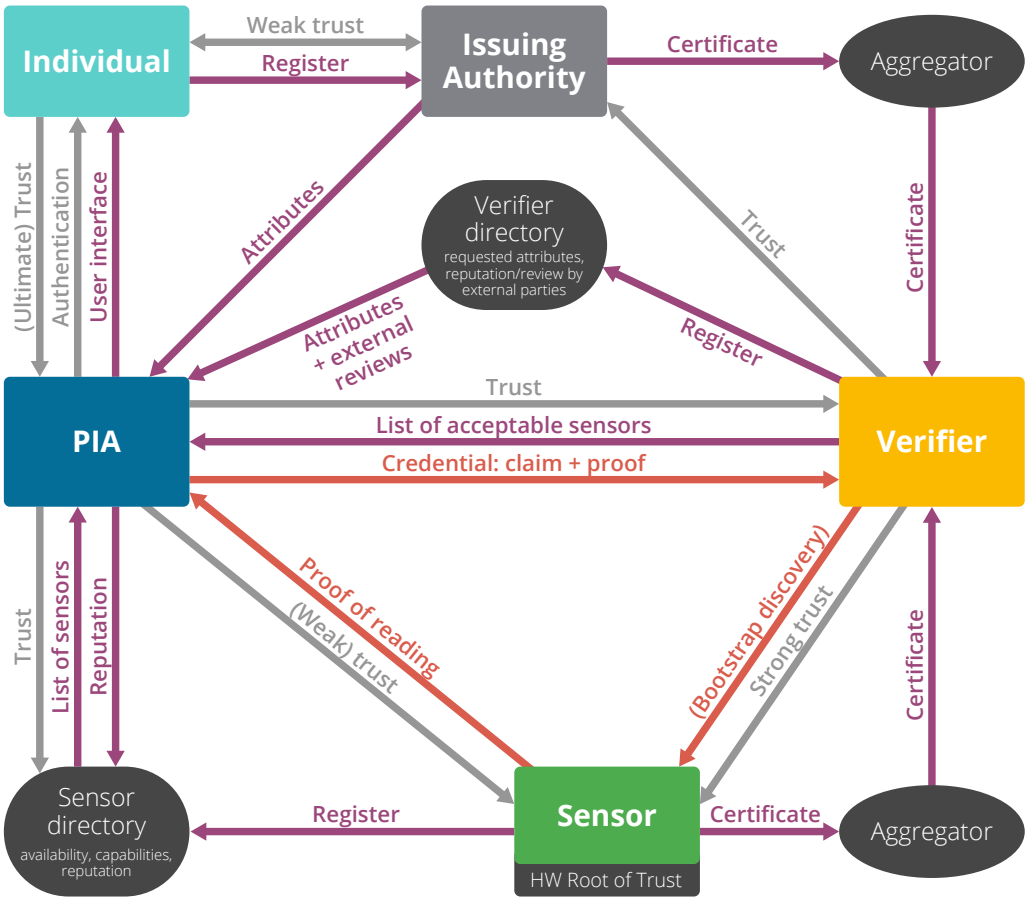

Figure 2.1: The network architecture proposed by the Digidow project [91].

identity information (*attributes*) about a single individual. Such attributes can be general information like name, age, or gender, biometric measurements, or specific data like a user's club membership ID. The purpose of the PIA is to store all of these attributes and share them with other parts of the Digidow network, if the owner wants it to do so. Ideally, all PIAs would be operated on personal devices by their owners to ensure that nobody gains access to them, but for a majority of the population this is likely not going to be an option. Providing these users with maximum control over their data therefore means giving them the freedom to delegate the task of operating their PIA to a third party they trust.

Sensors serve as points of interaction between individuals in the physical world and their PIAs in the digital world. Sensors can be cameras, fingerprint readers, or anything that can physically extract a biometric template for a physically present person. Within the Digidow architecture, a sensor exists solely to provide a PIA with a proof that the individual represented by the PIA has interacted with the sensor. Sensors can be operated by anyone because the decentralized nature of the Digidow architecture lacks a central authority that could grant or revoke the permission to operate a sensor. This means that PIAs must not trust sensors by default, but at the same time a sensor must be trusted to handle the biometric information of individuals in order to fulfill its function. The proposed approach to solve this conflict is a hardware root of trust that proves that the sensor is only operating a publicly known and trusted software version that does not store or forward the recorded biometric information in any way. By verifying this hardware root of trust, PIAs can trust that sensors are not malicious, even if they know nothing about the operator of the sensor.

The verifier represents the service that actually requires attributes about an individual. A verifier could be everything from a government checking digital passports at border crossings to a private homeowner who wants to automatically unlock the front door when they return home from work. Verifiers typically ask for attributes about an individual and take decisions based on them. Digidow puts no limitations on the attributes requested by the verifier, nor does it make any assumptions on how the received information is used afterwards. The key objective is to grant users (via their PIAs) agency over which information is shared with every verifier. What verifiers do with this information afterwards is out of scope for this project.

A simple Digidow transaction for verifying passports before opening automatic gates at a border crossing would look like this: The sensor detects the biometric features of an individual standing in front of the gate and provides the PIA with a digitally signed proof that the individual with those biometric features is currently located in front of the sensor. Afterwards, the PIA contacts the verifier to find out which attributes the verifier needs to allow the individual to pass the border crossing. For some countries, a proof of citizenship might be sufficient, others might ask for additional information like name, date of birth, vaccination status, or insurance coverage. The decision which attributes to request is completely left to the individual verifier. The PIA can then decide if it is willing to provide those attributes to the verifier or not. If it decides to do so, it combines the proof received from the sensor with proofs about the requested attributes and forwards everything to the verifier. Before the verifier can allow the individual to pass the border crossing, it has to verify if the received attributes are cryptographically valid and signed by trusted entities. Only then, the verifier can open the gate and allow an individual to pass.

The fourth key players needed within the Digidow architecture are *issuing authorities*. Their main task is to link attributes to biometric features, similarly to how sensors assign temporary locations to a biometric measurement. Governments or companies are typical examples of issuing authorities, but private

individuals might also issue attributes like permission to enter a house. This enables a PIA to prove for example that the individual with biometric features A is in front of the gate according to sensor X and that the individual with those biometric features is a citizen of Austria according to the Austrian government. If the verifier trusts both the sensor and issuing authority and the digital signatures are all valid, it can be certain that the person in front of the gate is an Austrian citizen. The important thing to mention here is that issuing authorities are not involved in Digidow transactions, making it impossible for them to learn when and to whom the attributes provided by them are presented. This is important because it protects user's privacy against the issuing authorities that provide attributes.

In order to allow verifiers to obtain up-to-date information on which certificates are used by various issuing authorities and sensors, there will be a need for central aggregators. Otherwise verifiers would have to contact every issuing authority directly to learn about its certificates, and more importantly its certification revocation list. Fortunately, distribution of certificate revocation information is not a new problem [23], so it will not be discussed any further in this thesis.

The final components of the Digidow architecture are the verifier directory and the sensor directory. The verifier directory was introduced to help PIAs decide whether they should provide attributes to a verifier or not. The idea is that nonprofit organizations or governmental privacy protection agencies publish lists of verifiers they know along with the attributes they are allowed to request. Those reviews can individually challenge verifiers to justify their requested attributes and their intended uses. PIAs could use those lists to make good decisions on behalf of users who do not want to manually decide about which data they want to share. The verifier directory is an optional component because Digidow transactions could also be conducted without it. Its main purpose is to reduce the amount of user interaction required by the PIA to make good decisions in the interest of its owner.

The inclusion of the sensor directory into the Digidow architecture is one of the contributions presented in this thesis. Therefore, it will not be discussed at this point, as the motivation for including it will be covered in substantial detail in section 3.2.

2.2 Network Anonymization

The fact that many digital applications are vulnerable to traffic analysis has long been recognized [17]. Unfortunately, all approaches put forward and discussed in this section do share the same issue. To obscure who is communicating with whom on a network level, communication cannot be transmitted directly between the communication partners and must instead be relayed by proxy devices. Indirect communication via such relays adds additional latency and bandwidth consumption to network connections, resulting in a constant trade-off between network performance and network anonymity. This section will discuss the various concepts for network anonymity that have been proposed in the past and evaluate how suited they are to achieve the research objectives documented in section 1.2.

2.2.1 Mix Networks

The first documented proposal to achieve network privacy for a distributed system was put forth by David Chaum [17] in 1981, when he suggested a concept for untraceable email communication. The core concept he proposed was a *mix*, a device that accepts messages from other clients, aggregating messages in batches, sorting them in lexicographic order and then forwarding them. To prevent both passive attackers on the network and the mix itself from learning anything about the messages, they are encrypted twice. First, the intended message is encrypted with the public key of the intended recipient and that encrypted message along with the address of the recipient is then encrypted again with the public key of the mix. Lastly, a mix must never forward the same message more than once because that would allow an observer to link multiple inputs to multiple outputs. A mix that operates in this fashion is often referred to as a *Chaum Mixer*.

Mix networks consist of multiple Chaum Mixers that allow clients to cascade their communication via multiple mixers, enabling them to control the trade-off between anonymity and network speed. Every additional mixer makes it harder for an attacker to find out where traffic is going, but at the same time it increases the time needed for a message to arrive at its destination. The biggest advantage of mix networks is that passive adversaries cannot find out who is communicating with whom, even if they have access to 100 % of the networks traffic. However, this requires Chaum Mixers to aggregate sufficiently large batches before forwarding and waiting for those batches to grow introduces massive and unpredictable network latency. There has been continued work on how to improve the latency of Mix networks [65] but no mix network has ever achieved privacy against passive adversaries with access to all network traffic without forbidding mixes from forwarding data immediately [2]. Nevertheless, there have been continued efforts to improve mix networks, mostly for applications like messaging that are less impacted by additional delays [2, 24, 79, 107, 130]. It should be emphasized however, that only one of those attempts (Loopix [107]) has seen any use beyond research prototyping. Recent efforts surrounding mix networks that might be adopted in the future extend the Loopix architecture [26] and focus on blockchains and cryptocurrencies [55], as those technologies are distributed and privacy sensitive. Thanks to the fact that blocks within a blockchain are created slowly and users already have to wait for new blocks to be created before a transaction is confirmed, the additional delays introduced by a mix network might be acceptable there.

After careful consideration, we came to the conclusion that it is currently infeasible to use a mix network based networking scheme to provide network privacy in a distributed digital identity system. One main argument for this conclusion is the added latency that would have a significant negative impact on user privacy. The other critical argument is that there are very few established mix networks available to build on and the ones that exist are both too small to provide privacy and not yet sufficiently tested to earn the necessary trust. Until those two things change, digital identity systems will have to use alternative strategies to obtain network privacy.

2.3 Onion Routing

The concept of onion routing [43] takes the idea of cascading multiple Chaum mixes and modifies it to be protocol agnostic. This is achieved by two key modifications: First, onion routing uses real-time Chaum mixes, which do not ag-

gregate packages in batches or send them in predefined intervals. Instead, every incoming packet is forwarded as soon as possible, just like it would be without an anonymization network. Second, the information about where to forward a packet is no longer included in every encrypted packet. Instead, onion routing connections undergo three different phases during their existence. In the first phase, an onion routing connection is established. To do so the client establishes a TCP connection to the first mix and tells it that this is a new onion connection. Then the client selects a second mix and instructs the first mix to establish a TCP connection to the second mix and informs it that it is now part of a new onion connection. This continues until the client is satisfied with the number of mixes between itself and its destination. After the connection has been established, the next phase sees the client communicate with its destination by encrypting the packets it would usually send with the keys of all the mixes involved in the connection. Every mix receives the packet on a specific TCP connection, decrypts the packet with its key and then forwards it to the next mix it still knows from the setup phase. The final mix ends up with the decrypted packet that contains the normal networking information that would have usually been used to deliver the packet directly, so it can just make sure that the packet reaches its intended target. If the final mix receives a response to a request, it adds layers of encryption for every mix and forwards the encrypted response to the next mix in the path.

To improve performance and prevent attackers from recognizing used keys, public key encryption is only used during the building stage of an onion routing connection. During this stage, temporary symmetric keys are negotiated that are used for encryption and decryption during the data transfer phase. Those keys are different for each direction because the encryption keys for the reverse path should only be known by the final mix, while only the client should have the keys for the other direction. The final stage of a connection is the tear-down where mixes know that no further packages should be processed and the temporary keys can be deleted. This also happens if the TCP connections between the client and the first mix or two of the mixes break down and cannot be recovered.

Onion routing has two main advantages over mix networks: First, it supports arbitrary protocols because a constant bi-directional stream of data becomes possible. This allows existing network applications to use this concept directly by treating the entire onion routed connection just like they would treat a regular proxy server (e.g. SOCKS). Second, onion routing significantly improves the performance of connections by reducing the number of required public key cryptography operations and maintaining open connections between the mixes of active connections. Unsurprisingly, the trade-off between network performance and network anonymity means that these gains in performance have to be paid for by a loss of network privacy. Onion routing is forced to change their threat model from a passive adversary that has access to the entire network to a passive adversary that has only access to a fraction of the network [34]. This can be directly tied to the use of real-time mixes instead of regular ones. A real-time mix forwards traffic immediately, allowing onion routing to forward any type of protocol because the sequence and order of messages is maintained, but at the same time attackers can conduct timing based traffic correlation attacks. As a general rule, it can be said that onion routing does not protect against attackers that can see the traffic before it enters the first mix and after it leaves the last mix [9, 85, 95, 114].

This raises the question whether a digital identity system could still be considered privacy preserving if it decides to rely on onion routing to anonymize its traffic. To the best of our knowledge, there is no single entity with access to 100 % of the Internet's traffic, but there are entities that control multiple

autonomous systems (AS) or internet exchange points(IXP), enabling them to observe a substantial share of Internet traffic. Johnson et al. [66] have shown that the organizations behind large ASs or IXPs, like DE-CIX or Level 3 Communications could successfully deanonymize users over a longer period of time, while every single connection only has a probability of about 1% of being deanonymized. This also corresponds with the information leaked by Edward Snowden regarding the Tor network [34], the most popular currently available onion routing network. According to those leaks, intelligence agencies are capable of deanonymizing some of Tor's users some of the time, but have not managed to selectively target specific Tor users. Based on the assumption that their capabilities have not substantially changed since then, the question arises whether occasional random deanonymization of network traffic is acceptable for a privacy preserving identity system. Our conclusion is that this is acceptable because using digital identities in the physical world will always leave traces that external entities can observe. Even if the network would provide perfect anonymity, physical surveillance of a sensor would still reveal who is interacting with it at which times. As long as individuals cannot be targeted specifically and there is not enough information leaked to automatically build profiles on all users over time, the primary requirements of a privacy preserving digital identity system are still fulfilled. Therefore, we believe that onion routing is the best suited available technology to achieve network privacy for distributed systems and will be used for the remainder of this thesis.

With the decision for onion routing made, the next task is to identify an existing onion routing network that would be best suited to support a distributed digital identity system. While there are ideas to integrate onion routing into the infrastructure of the network directly [18], current implementations of onion routing are mostly available as overlay networks on top of the established Internet infrastructure. The two largest existing networks will be considered as potential foundations for a distributed digital identity system.

2.3.1 I2P

The I2P project [61] seems like the perfect fit for the requirements of this thesis. It was designed to provide a peer-to-peer network that uses onion routing to allow all peers to communicate with each other. Connecting to devices that are not part of the I2P network is not supported, but also not needed for a distributed system. Devices connected to the I2P network are referred to as I2P *routers* and receive their own router identifier that can be used to contact them later on [7]. A key difference to usual onion routing is that I2P does not support bi-directional connections. Every established connection (I2P *tunnel*) only operates in one direction, meaning that two connections must be established for bi-directional communication.

The I2P network is fully decentralized, meaning that there are no central authorities that have to be contacted before joining the network. There is one catch however, a new I2P client needs to know about at least one existing I2P client to obtain some information about the network. Since this can be any active I2P router, this does not lead to centralization in theory but in practice there exist a few long-lived I2P routers (*reseed* servers) that are typically used for this purpose. The information regarding the entire network is stored in the *netDB*, a distributed hash table maintained by all active I2P routers, so no router ever has full knowledge about the network [56]. This netDB contains information about every available I2P router, including their router id and their hostname/ip address. Additionally, the netDB also contains information about every service available within the network. This is necessary because I2P needs to

conceal which router is providing a service, so services actually have to establish a tunnel to a random router and then publish service information that tells potential clients that they can reach the service by connecting to that router and asking it to forward the message to the final destination. Effectively, every I2P tunnel consists of two halves. The first half are routers chosen by the sender and the second half are routers chosen by the receiver. This ensures that both parties can be certain that their privacy is protected.

The I2P network enables clients to communicate with each other via onion routing, which is the primary requirement for a distributed system that cares about network privacy. So, from a functional perspective, I2P is well suited to support a privacy-preserving distributed system. However, functional requirements are not everything, so the current capabilities of the I2P network must be analyzed next to find out if the I2P network could handle the load associated with a global distributed system. While it is not trivial to measure a fully decentralized network like I2P, there are ways to obtain statistically significant measurements by operating multiple I2P routers and aggregating and extrapolating their data [56]. This tells us that the I2P network currently has around 27.000 active routers with about 16.000 different IP addresses. Even fewer, only about 12.000 are actually reachable, the rest is most likely blocked by firewalls or NATs, so the concept of every I2P router forwarding traffic for others does not hold. The situation becomes even worse, when analyzing the advertised bandwidth provided by I2P routers. Less than 3000 of them share more than 2 MB/s of bandwidth, while more than 15.000 share less than 48KB/s. Most applications have bandwidth requirements that are much higher than this and more importantly this limited bandwidth capability increases the probability of congestion at I2P routers leading to additional latency. This problem is intensified by the heavily decentralized setup of I2P. Clients only ever see a fraction of the netDB, meaning they cannot simply choose the post powerful routers within the network, they have to use what they see and there is a high chance that some clients will end up with only routers providing less than 48KB/s of bandwidth, which is insufficient for anything except the most basic of communication.

This leads us to conclude that I2P is an excellent candidate to support privacy preserving distributed identity system in theory, but it has not yet gathered a large enough user base that is willing to contribute routing capacity and internet bandwidth to support larger applications. However, this decision is purely due to the current state of the I2P network and that could easily change in the future.

2.3.2 Tor

The Tor project [34, 123] is probably the most commonly known tool when it comes to concealing network location. Although it was suggested a year after I2P, Tor managed to capture a significant user base by focusing on a scenario not supported by I2P: Establishing anonymous connections to services that are available on the Internet. This was an attractive offer for users who wanted more privacy with their current Internet activities and gave users living in oppressive countries with censorship a safe way of connecting to the internet.

Tor can be considered as a reference implementation for onion routing with the acronym Tor being short for “The Onion Router” and one of the inventors of onion routing helping with the design of Tor. The key feature of Tor are the so called *exit nodes* that allow users of the Tor network to establish outgoing connections to any device on the internet. Another important difference in comparison to I2P is that Tor does not expect every client to also act as a mix.

This means that it is much harder for an observer to find out if someone is using Tor. The only way to find out is to monitor the client's web traffic and check if any of its outgoing connections are going to a Tor relay and Tor's efforts to circumvent censorship [120] make this even harder to do.

While the Tor project originally did not support accepting incoming connections via the Tor network, support for this feature was later introduced [126] and continuously developed since then [127]. Clearly inspired by I2P, this allows Tor clients to select a random node within the Tor network as entry point for its service and publish that information in a distributed hash table. This enables Tor clients to accept incoming connections via Tor, which in turn allows direct peer-to-peer connections between clients, just like I2P does. There remain two key differences that should be pointed out at this stage: First, the Tor network is not fully decentralized. It depends on a set of manually selected directory authorities that control who is allowed to join the network. This does not mean that they control which clients are allowed to use the network, but they do control which mixes clients will use. The need to trust these directory authorities necessitates a high level of trust that can only obtained by verifying that they are acting as expected. Chapter 6 will provide a detailed analysis of the behavior of Tor's directory authorities. Second, every Tor client downloads the entire list of available Tor mixes, while I2P routers only know about a small fraction of the network. This puts limitations on how large the Tor network can grow before Tor clients can no longer process the entire list of available Tor mixes, but has no real impact from a privacy perspective.

This directly leads to the question of feasibility and for that an estimate on the current size and capabilities of the Tor network is needed. Fortunately, the Tor project does provide extensive statistics about itself that can be drawn upon to answer these questions [86, 125]. The Tor network currently consists of about 6200 different mixes that distribute traffic for other users. More than 4000 of them provide a bandwidth of more than 2MB/s and only a few hundred provide a bandwidth of less than 50KB/s. At the same time, the Tor network sees more than 3 millions users per day that consume about 250Gb/s of bandwidth. Although the Tor network is smaller than the I2P network, it services more than 100 times more users than the I2P network. This is mainly possible because the available mixes all volunteered to forward traffic and allocate the necessary resources and bandwidth to execute this task efficiently. The substantially larger user base has also attracted significantly more attention from research. There is ample research available on every aspect of the Tor network. Potential attacks [11, 19, 20, 95, 114], provided privacy guarantees [66, 102], and real world usage [86, 102] have been the subject of extensive research that drove the Tor project to continuously improve their network. Beyond research that mostly aimed to strengthen the Tor network, there have also been several documented cases of powerful adversaries actively compromising the network [8, 27, 30, 98] including both the American FBI and the British GCHQ. One could argue that a network under constant attack is not the best choice to provide security, but this argument would be misleading. With every detected attack on the Tor network, its security was improved to address the flaw that enabled the attack, making the Tor network more resilient to future attacks. While this is no guarantee that there are no undiscovered security flaws within the Tor network, the knowledge of all the attacks that failed and the sheer amount of research that has tried to attack it make the Tor network the most trustworthy anonymization network currently available.

All those reasons combined lead to the inevitable conclusion that any privacy preserving digital identity system deployed in the near future should rely on the Tor network to achieve network privacy. It has enough other users and traffic to blend in with, it has the most data transfer capacities and most importantly

it has stood the test of time even when attacked by powerful adversaries, making it very likely that it will also be able to protect the privacy of its users in the future. Since the design details of the Tor network are instrumental for the contributions presented in chapter 4,5,6 and 7, section 2.4 provides a detailed description of how the Tor network functions.

2.4 The Tor Network

The inner workings have been originally published as a paper [34] and the current specification with all the changes implemented over time is also publicly available [33], making it easy to understand the inner workings of the Tor project without having to read the source code, which is also open source. The Tor network refers to its mixes as Tor *relays*. Those relays are operated by volunteers, so everyone can contribute to the Tor network by adding more relays to it. Connections through the Tor network are called *circuits* and typically include three Tor relays: The first relay allows a client to enter the network and is commonly referred to as *guard* relay, the second relay that purely forwards the data from the guard, usually referred to as *middle* relay, and finally the *exit* relay that forwards requests to their final destination. Figure 2.2 visualizes such a typical Tor connection. Technically, Tor clients could choose to build circuits that consist of a different number of relays, but it is not easy or recommended to so because it makes Tor traffic more vulnerable to certain attacks [14]. In order to establish connections through the Tor network, clients need to obtain a list of currently available relays and their capabilities because not every Tor relay can be used as a *guard* or *exit* node. *Exit* nodes appear like the origin of traffic originating from the Tor network, forcing their operators to deal with all kinds of abuse complaints linked to problematic behavior via the Tor network. *Guard* relays are the entry point into the Tor network and therefore the only devices that know the network location of Tor clients. This makes them interesting points of attack because the anonymity of the user is compromised if a Tor client can be tricked into choosing a not trustworthy guard relay. To minimize the risk of that happening, a Tor client keeps using the same *guard* relays for all circuits to minimize the number of Tor relays that know its identity [32]. For a long time Tor clients only used a single guard [29], but this was recently changed to two guards to better distribute load across the Tor network [31].

The process of obtaining the list of currently active Tor relays is commonly referred to as *bootstrapping* and requires new clients to connect to an already running Tor relay to learn about the current network. A random set of stable Tor relays is hardcoded in every Tor release to make this procedure as easy as possible. The list of all currently available Tor relays is called the *consensus* and is arguably the most important piece of information for every Tor client. The consensus document includes address and key information about every relay, provides information about the capabilities of every Tor relay as well as their stability and available bandwidth. For performance reasons, not all information about relays is included in the consensus. Tor's directory specification [124] specifies several *descriptor* formats that provide advanced information about a specific relay. In order to request any of those descriptors, the Tor client needs a valid consensus first and can then request more specific descriptors about the relays it wants to use. A new consensus is created every hour and usually valid for up to three hours, so there are multiple valid consensuses at most times which gives clients more freedom when they update their consensus information. Section 2.4.1 describes the process of creating and verifying the consensus document in more detail.

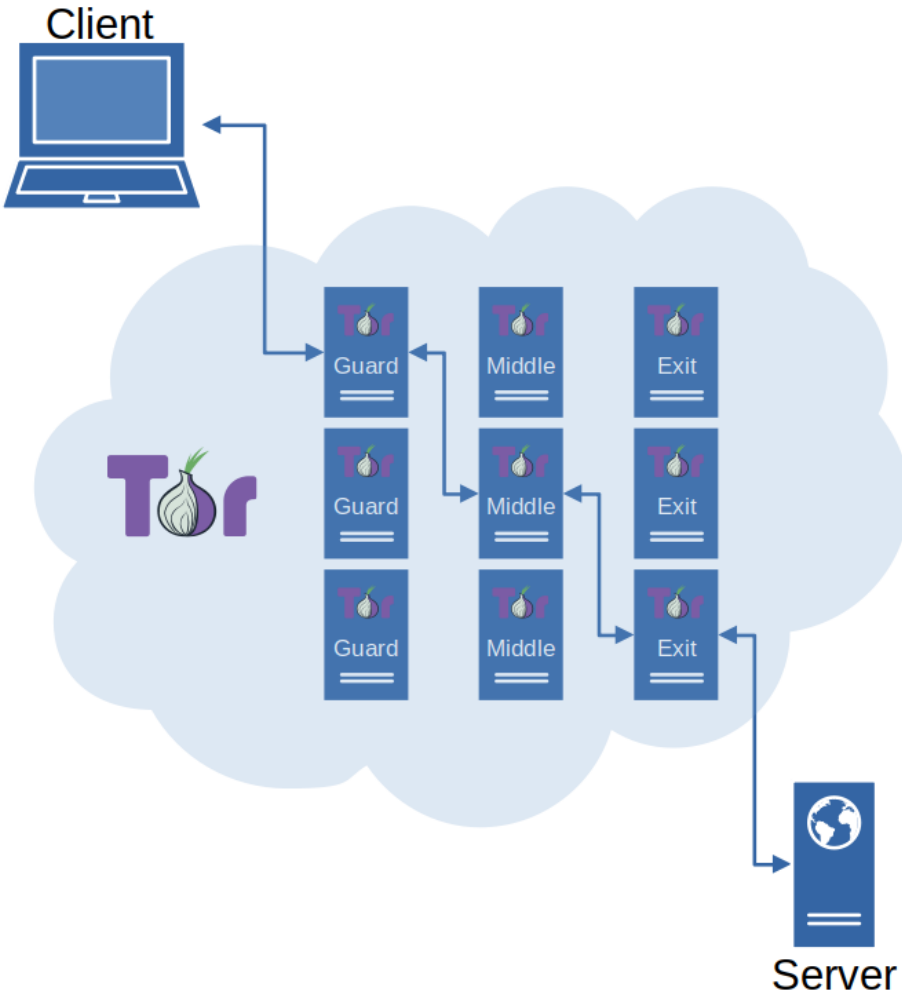

Figure 2.2: How regular Tor connections work

Onion routing in general and Tor specifically were designed in a way that supports arbitrary protocols by being able to handle arbitrary network streams. Technically, Tor can only handle TCP streams but this is sufficient for most real world applications. When running on a client, not as a relay, Tor presents itself as a SOCKS5 proxy [81] to other applications. This means that almost every network application that supports using a proxy server can be configured to communicate via Tor. The most common applications used in combination with Tor are web browsers;so common in fact that the Tor project has begun to maintain a modified version of the Firefox[9] browser that removes some features that might compromise a user's privacy and redirects all communication via Tor by default. This Tor Browser[10] is the recommended and most common way for users to use the Tor network.

However, some applications require advanced control over the behavior of a local Tor client. In order to support this kind of functionality, the Tor control protocol [122] supports fine grained control over almost every aspect of the Tor application. It can be used to create/modify/destroy circuits, it can make arbitrary configuration changes, it can obtain arbitrary logging information, or even restart/shutdown Tor entirely. The Tor Browser for example uses the control protocol to find out which countries traffic is currently being routed through and display this information to the user. Other applications like OnionBalance or OnionShare[11] would not be possible at all without the control protocol.

2.4.1 Directory Authorities

The decision about which relays are included in the Tor network is a decision of great importance for all Tor users. If malicious relays are included, their anonymity might be compromised. If too few relays are included, the network might not be able to handle enough traffic and the network becomes useless. Besides, the entity responsible for these decisions might be forced by authorities to provide specific users with modified variants of the consensus that only include relays under the control of an attacker. To mitigate this risk, the responsibility for creating the Tor consensus is not given to a single entity. A small group of trusted relay operators that are known supporters of the Tor project and spread across multiple countries have been chosen to fulfill this task collaboratively. Their relays are also known as *directory authorities* and information about them is hardcoded within the Tor application. At the moment, there are 9 directory authorities (*moria1*(USA), *bastet*(USA), *longclaw*(Canada), *Faravahar*(USA), *dizum*(Netherlands), *gabelmoo*(Germany), *tor26*(Austria), *dannenberg*(Germany), and *maatuska*(Sweden)).

Every directory authority maintains an independent view on the Tor network and publishes its perspective hourly in a *network status vote*. The consensus is also created hourly by every directory authority but this is not directly influenced by their personal view on the network. Instead, they collect the network status votes from all available directory authorities (including their own) and include everything in the consensus that is included in a majority of votes. So a relay is only included in the consensus if more than 50 % of the votes include the relay. This also applies to properties of a relay, so a relay is only considered fast if more than 50 % of the votes believe it to be fast. If everything works as intended, all nine directory authorities have access to all nine votes and produce the same consensus, which they sign digitally before publishing it. By exchanging signatures, each directory authority ends up with a consensus docu-

[9] https://www.mozilla.org/en-US/firefox/
[10] https://www.torproject.org/download/
[11] https://onionshare.org/

ment that has been signed by all directory authorities. Note that while an ideal consensus is built from nine votes and has nine signatures, a valid consensus only requires the signatures of a majority of voting directory authorities. If all nine authorities are voting, five signatures are needed, but if two authorities stop voting there would only be seven voting authorities and four signatures would be enough to sign a valid consensus.

Tor Flags

The Tor network describes the properties of Tor relays with a series of flags that are assigned by the directory authorities if the relays meet the necessary criteria. The following list provides a selection of the more important flags currently present in the Tor consensus:

- *Valid:* Assigned if the version of Tor run by the relay is not known to be broken. Invalid relays are not included in the consensus.
- *Running:* Assigned to all running relays. Requires the directory authority to be able to connect to the relay. Relays that are not running are not included in the consensus.
- *V2Dir:* Assigned if the relay supports the V2 directory protocol. Unless actively disabled, all current Tor versions obtain this flag.
- *Fast:* Assigned if the relay is suited for high bandwidth ($\geq$ 105 KB/s) connections.
- *Stable:* Assigned if the relay is suited for long-lived connections. Requires the relay to have a mean-time-between-failure of more than seven days.
- *HSDir:* The relay is part of the hidden service directory. Assigned only if the relay is stable, fast, and has been up for more than 96 hours.
- *Guard:* Assigned if a relay is suited to be the first node of a Tor connection. Requires a relay to be fast, stable, be a V2Dir, have an at least median uptime, be at least a few weeks old, and have a bandwidth of more than 2 MB/s.
- *BadExit*: Assigned if a directory authority believes that an exit relay should not be used by clients. This flag is unusual because there is no specification on how that assumption should be built, the only example given is using an internet provider that is known to block/censor traffic. In practice, the assignment process for this flag is semi-manual.

Tor Bandwidth Authorities

While it is easy for a directory authority to keep track of the uptime of relays because relays have to upload new descriptors regularly, measuring their bandwidth is more challenging yet still important. Relay operators usually think of bandwidth in terms of what they pay their internet provider for and make a fraction of that bandwidth available to the Tor network. But the actually available bandwidth does not always correspond with what clients pay for, leading to Tor relays advertising more bandwidth than they can actually handle. Even worse, malicious relays can advertise huge amounts of bandwidth that they could never handle, just to cause the Tor network to send them lots of traffic that they will drop. Both cases result in a deteriorating user experience for all Tor users because of incorrect bandwidth information.

To address this issue, Tor uses several bandwidth authorities [68] which are responsible for measuring the available bandwidth of relays. Since the results

of these bandwidth measurements must be incorporated into the votes for the consensus, only directory authorities can be bandwidth authorities. This means that for every consensus vote, some authorities make bandwidth decisions based on advertised bandwidth, while others decide based on their measurements. To prevent malicious relays from only responding to measurement traffic, these measurements must also take place via the Tor network in order to appear just like regular traffic. This means that every bandwidth measurement is going via several nodes, making it hard to tell for certain if the measured relay really was the bottleneck during the measurement. Currently, Tor has two different algorithms for measuring relay bandwidth in use (torflow[12] and sbws[13]), resulting in three different ways how a directory authority can determine bandwidth information about relays.

2.5 Tor Onion Services

Onion services (formerly known as hidden services) were originally conceived as an experiment to learn more about what kind of tasks the Tor network could be used for [47] and have found significant user acceptance since then. Onion services enable users of the Tor network to accept incoming network connections from other Tor users, a scenario that was originally not supported by the Tor project. At the time of writing, there were more than half a million onion services deployed within the Tor network responsible for about 10 Gb/s of traffic making up about 4 % of the entire Tor network's traffic [125].

Interestingly, anonymous communication within the Tor network is actually more complex than anonymous communication with a device outside of the Tor network. The main reason for this increased complexity is that regular Tor connections only protect the privacy of one client, while onion services have to protect the privacy of both communication partners. Considering that I2P published their solution to this problem one year before the Tor project announced their support for onion services, it is not surprising that the functionality of onion services has a lot in common with how I2P enables anonymous internal communication. Onion services have also been a constant target for attackers, forcing the Tor network to regularly update their implementation and occasionally even the specification. Therefore, there were two variants of onion services deployed within the Tor network for a long time: version 2 onion services [126] (since 2005) and version 3 onion services [127] (since 2018). Version 2 was deprecated in 2021 [47], so everything in this thesis—unless otherwise specified—refers to the currently supported version 3 onion services.

Figure 2.3 visualizes the complex procedure that was designed to create and communicate with onion services using the best privacy protection achievable with the Tor network. In total, six different types of circuits across at least 15 different Tor relays are needed with at least 5 different public/private key pairs and additional symmetric keys. The functionality provided by onion services can roughly be split in three parts: The setup required by Tor clients that want to run an onion service, the service provided by the Tor network to support the distribution of onion service information, and finally the actions required by the client to connect to a running onion service. All of those will be described in more detail in the following sections:

[12] https://gitweb.torproject.org/torflow.git/tree/NetworkScanners/BwAuthority/
[13] https://gitlab.torproject.org/tpo/network-health/sbws

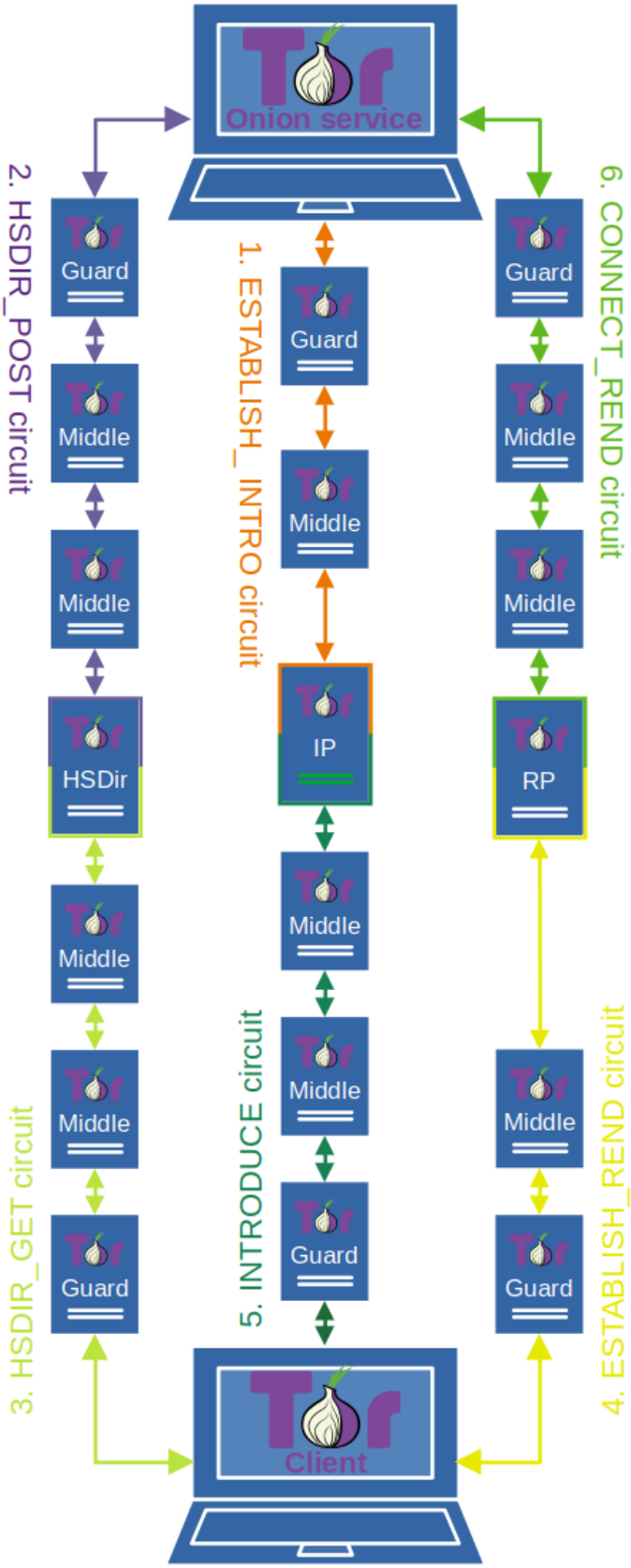

Figure 2.3: How Tor onion services operate

2.5.1 Creating a New Onion Service

The first thing every onion service needs is an elliptic curve (ed25519) key pair that grants control over an onion service. The private part of that key is referred to as *MASTER KEY* and the public part is called the *AUTH KEY*. Every cryptographic proof that ensures the integrity of onion service operations ties back to the *MASTER KEY* and requires the *AUTH KEY* to be verified. The *AUTH KEY* is encoded in the *onion address* that serves as hostname and identifier for an onion service.

After the main key pair is ready, a Tor client intending to run an onion service needs to provide other Tor clients with a way to connect to it, without disclosing its identity. This is achieved by randomly selecting relays within the Tor network as *introduction points* (*IP*). The client establishes *ESTABLISH_INTRO* circuits to these introduction points and asks them to act as introduction points for their new service. According to the protocol [127], Tor relays are not allowed to refuse such a request unless it is invalid. By default, every onion service selects three different introduction points, but any number between 0 and 20 would be valid. Every introduction point receives a fresh key (the *INTRO AUTH KEY*) that can be later used to link incoming requests to the correct onion service. The circuit to the introduction point has to be kept open continuously by the onion service because this circuit is the only possible path of communication between the introduction point and the onion service. If this circuit fails, the introduction point becomes unusable; if all circuits to introduction points fail, the onion service becomes unreachable. Before clients can start connecting to the service however, they need a way to learn about the introduction points chosen by the onion service. To achieve this, the future onion service bundles the information about all its introduction points in one file called *service descriptor*.

Figure 2.4 visualizes the layout of such a service descriptor along with all the data it provides to a Tor client. However, the only thing distributed to clients is the static onion address of an onion service, not the service descriptor that has to be changed every time the circuit to an introduction point fails. This means that Tor clients need a reliable way to request the currently valid service descriptor for an onion address. This service is provided by the *hidden service directory* (*HSDir*).

2.5.2 The Hidden Service Directory

The hidden service directory is a distributed hash table that contains information about every currently available onion service and its current service descriptor. It is also arguably the aspect of Tor onion services that changed the most between the onion service versions 2 and 3, after researchers demonstrated that the implementation in version 2 allowed malicious attackers who joined the hidden service directory to collect information about every currently deployed onion service, restrict access to selected onion services, and count the number of Tor users that connect to them [11, 102]. This forced the Tor project to completely redesign the hidden service directory for version 3 to defend against malicious relays within the hidden service directory.

The HSDir is formed by all Tor relays that fulfill the requirements to obtain the HSDir flag in the current consensus. By comparing the requirements for the HSDIR flag and the V2Dir flag in section 2.4.1 one can see that this is the first aspect that saw significant change between versions 2 and 3. Since all running Tor relays need to know the current consensus, they are always aware of which

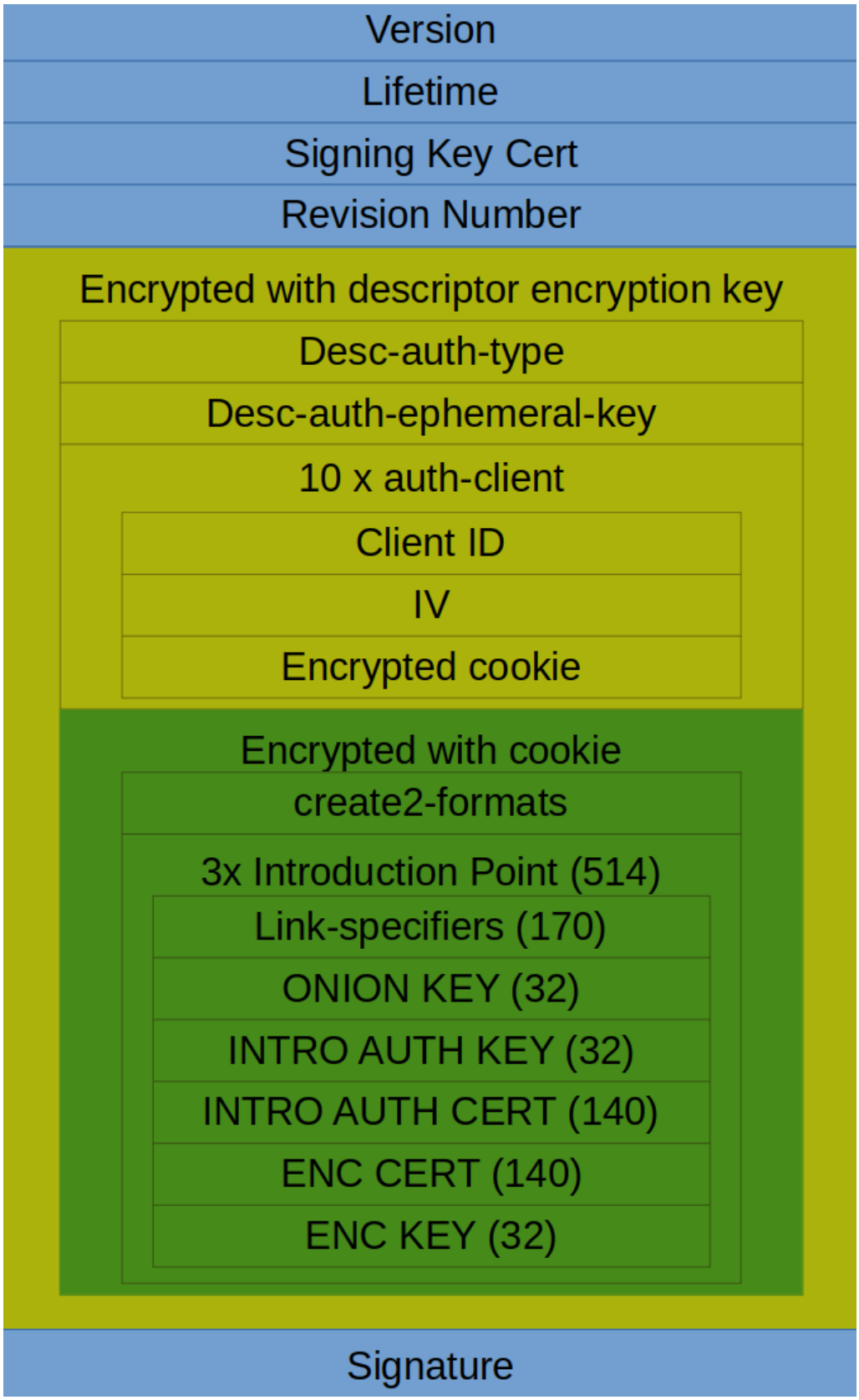

Figure 2.4: Structure diagram of a service descriptor

relays currently form the hidden service directory. The next decision needed is which relays within the HSDir should be responsible for which service descriptors. To answer this question, a *HSDIR INDEX* is calculated for every relay that depends on the identifier of the relay, the current date and time, and a shared random value (*SRV*) included in every consensus. This SRV is changed every 24 hours and makes it impossible for attackers to predict in advance which member of the HSDir will be responsible for a specific service in the future. It was introduced in version 3 to prevent attacks that relied on predicting the future layout of the HSDir [11]. The HSDir relays are sorted alphabetically based on their *HSDIR INDEX* with every relay being responsible for uploads whose hashes are alphabetically between its *HSDIR INDEX* and the index of the next HSDir relay. This means that the share of the hidden service directory handled by every relay is not equal and subject to change with every new shared random value.

Before onion services can publish their descriptors, they need to know the ID of their descriptor. In version 2, the onion address was used for this purpose, but that allowed malicious HSDir relays to harvest existing onion addresses [11, 102]. Version three solves this issue by deriving a new *blinded public key* (*BPK*) from the *AUTH KEY* This BPK changes periodically and does not leak any information about the *AUTH KEY* it was derived from. Further protection against malicious HSDir relays is achieved by encrypting the service descriptor with a symmetric key (*descriptor encryption key*) that is also derived from the *AUTH KEY*. This prevents attackers within the hidden service directory from learning onion addresses. Without knowledge of the onion addresses, even HSDir relays cannot read the service descriptors they are distributing. With the *HSDIR INDICES* and the blinded public key calculated and the service descriptor encrypted, an onion service can finally proceed to upload its descriptor into the hidden service directory. Since onion service operators cannot be expected to trust every single relay in the hidden service directory, every descriptor is not just uploaded to the relay responsible according to the *HSDIR INDEX*, but also to the three relays that come right behind it. Since those four uploads are still at four consecutive positions within the hidden service directory, a *replica* value is used to modify the BPK and determine a new second position within the hash ring of the hidden service directory, where the descriptor will also be uploaded to. This results in a total of 8 uploads for each service descriptor, which is still not the total number of uploads per onion service. As mentioned in section 2.4, there are multiple consensuses valid at every point in time. To prevent issues during times when the consensus is changing its SRV or transitions from one day to the next, every onion service keeps a second service descriptor that uses the previous day/SRV to support clients that are not fully synchronized with the onion service. This results in a total of 16 descriptor uploads per onion service. These uploads happen immediately after an onion service has been created. A running onion service re-uploads its service descriptors after between 60 and 120 minutes or if the service descriptor has changed.

Tor clients that want to download from the hidden service directory need to know the blinded public key of the service descriptor they are interested in. Typically, this information is extracted automatically from the onion address that encodes the *AUTH KEY*. With that information the client can calculate the *HSDIR INDICES* of all relays and identify the relays that should hold the desired descriptor. Note that onion services always upload to four consecutive HSDir relays, but clients always try one out of the first three. This guarantees that even if a relay within the hidden service directory goes offline, clients still have three relays available that hold the information they seek. Clients randomly decide if they try to download the main or the replica, which should distribute load fairly across the hidden service directory. If a download attempt fails, the client continues to try until it has exhausted the six HSDir relays that it is supposed

to download from. After a successful download, the client uses the *AUTH KEY* to derive the symmetric key needed to decrypt the service descriptor. The decrypted service descriptor is then stored in the internal cache of the Tor client, so future connections to the onion service will not require downloading the service descriptor again as long as the current one remains valid. With this information, the client is finally ready to establish a connection to the onion service.

2.5.3 Connecting to an Onion Service

While a Tor client is still downloading the service descriptor, it can simultaneously establish a circuit to a *rendezvous point* (*RP*). Just like introduction points, any Tor relays can be chosen by the client to act as rendezvous points. Once the circuit is ready, the Tor client informs the relay on the other side that it should act as rendezvous point for an onion service connection and provides it with a *RENDEZVOUS COOKIE* consisting of 20 random bytes. After the service descriptor is downloaded and the rendezvous point is ready, the Tor client creates a circuit to one of the introduction points specified within the service descriptor. Via this circuit, an *INTRODUCE1* cell is transmitted to the introduction point. The layout of this cell is shown in Figure 2.5 and can logically be split into two parts. The unencrypted part is intended to be parsed by the introduction point and the encrypted part—encrypted with the *ENC KEY* from the service descriptor—meant only for the onion service. The only important piece of information in the unencrypted part is the *INTRO AUTH KEY*. It is used by the introduction point to identify the circuit via which the encrypted part of the introduction cell should be forwarded. If the onion service has an active circuit associated with the *INTRO AUTH KEY* specified in the *INTRODUCE1* cell, it wraps the encrypted part of the cell in a new *INTRODUCE2* cell and forwards it to the onion service. If the introduction point cannot forward the cell, it responds to the client with an *INTRO_NACK* cell, otherwise it sends an *INTRO_ACK* cell to the client, after the *INTRODUCE2* cell was successfully sent.

When the onion service receives an *INTRODUCE2* cell, it checks if an identical *INTRODUCE2* cell has already been received before. This is necessary to prevent replay attacks. If the introduction request is valid, it uses the received link specifier to establish a new circuit to the rendezvous point chosen by the client. Once the circuit has been established, it uses the onion key specified by the client to prepare a handshake that sets up a shared symmetric key for end-to-end encrypted communication between the onion service and the client and sends a *RENDEZVOUS1* cell containing the *RENDEZVOUS COOKIE* and the handshake information to the rendezvous point. The rendezvous point uses the cookie to find out who the onion services wants to rendezvous with and logically connects the two circuits by forwarding everything arriving at one circuit to the other one. Finally, it forwards the handshake information from the onion service to the client, enabling the client to complete the handshake and start communicating with the onion service. The results of this process is a six-hop circuit between the onion service and the client, with three relays chosen by the client to protect its anonymity and three relays chosen by the server to do the same. However, Tor relays that are chosen freely by a Tor client—like the introduction point and the rendezvous point—do count as protecting the privacy of the Tor client that selected them. This is why—as seen in Figure 2.3—the circuit from the client to the rendezvous point uses the rendezvous point as exit node, while the onion service, which had no say in the selection of the rendezvous point adds a second middle node (technically, it is an exit node, but since it does not require the *Exit* flag, it is clearer to label it as a second middle node).

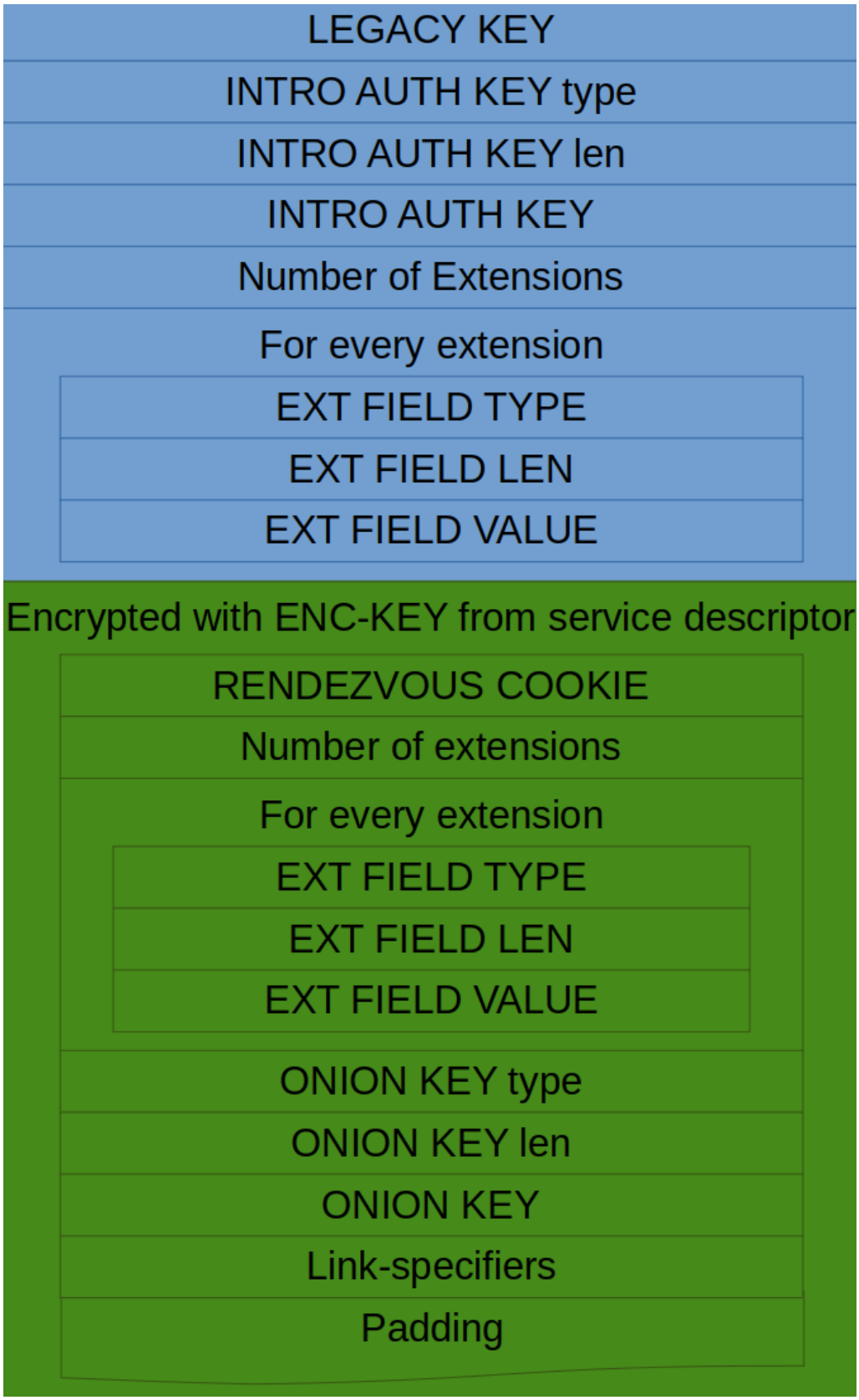

Figure 2.5: Structured contents of *INTRODUCE1* cell

2.6 Metadata resilient messaging

The biggest area where distributed systems on top of the Tor network have been implemented in practice is instant messaging. The Tor project itself initiated Tor Messenger [118], a tool meant to interact with various different providers of instant messaging (like Jabber, IRC, Google Talk, Facebook Chat, ...) via the Tor network. This effort had to be abandoned [117] because it added high latency to all communication without preventing the central servers of the messaging providers from monitoring their users.

More interest was gathered by fully decentralized messengers that tried to prevent tracking via metadata entirely by exchanging all network communication directly via onion services. The first implementation of this approach was TorChat [78]. It managed to gather some interest but failed to become a stable project and was abandoned fairly soon. In its place, the new project Ricochet [15] tried to establish a privacy preserving instant messenger and suffered the same fate as its predecessor. Despite those setbacks, there still seems to be a consensus that privacy preserving messaging is needed and new attempts to establish metadata resilient messaging continue to be made. At the time of writing this thesis, there were at least five different actively maintained projects that provide this functionality:

1. Cwtch [101]
2. Briar [111]
3. Speek! [128]
4. Ricochet Refresh [12]: An attempt to revive the discontinued Ricochet messenger.
5. OnionBalance [16]: Primarily a file sharing tool, but it also includes a peer-to-peer messaging functionality that uses Tor onion services.

This raises the question why centralized services like WhatsApp [76] or Signal [89] are still the standard when it comes to instant messaging. While a detailed analysis for this is out of scope for this thesis, we did identify four issues that all of these applications have in common that could cause users to not use them in practice.

The first issue concerns contact discovery. Popular centralized messengers are using phone numbers as contact identifiers, enabling them to automatically add all contacts that are already located in the users address book. A privacy preserving messenger cannot do this because its privacy partly depends on not publicly linking onion addresses to individuals. This forces users to manually add all their contacts again, a burden that many users are unwilling to accept. Distributed digital identity systems face the same issue when it comes to service discovery, which will be discussed in section 3.2.

The second issue concerns the additional network latency introduced by the Tor network. Instant messaging is often used as a synchronous communication form where users send messages and actively wait for the response. Adding a latency of several seconds to both the message and the response forces the user to wait much longer for the expected response, which negatively impacts user experience. Most messengers try to mitigate this issue by maintaining active connections to all contacts when they are online. This reduces the latency of individual messages because the needed circuits and rendezvous points are already established but it intensifies two other issues.

Every connection to another onion service requires a dedicated circuit with three different Tor relays. This always limits the amount of messages that a

messenger can send in parallel, but if connections to all clients are established all the time this limits the amount of total contacts a user can have. Even if there are still enough Tor relays available, it is important to consider that the Tor client will usually try to pick reliable and stable relays first, meaning that as more and more connections are established in parallel, the latency and reliability of circuits will deteriorate. This strongly incentivizes users with lots of contacts to stay on centralized communication platforms.

Finally, energy consumption must also be considered because most instant messaging is done from mobile devices that are powered by batteries. Research has shown that operating Tor onion services on mobile devices has a significant impact on bandwidth usage and energy consumption [72], even if they are hardly ever receiving incoming connections. Decentralized instant messaging that aggressively opens circuits to minimize latency comes with a high energy cost that is again unlikely to be tolerated by a majority of users.

All of these challenges also apply to decentralized digital identity systems and while some of the solutions proposed in this thesis are specifically tailored towards the Digidow architecture, others are more generic and could with slight modifications also be applied to the use case of decentralized instant messaging.

Chapter 3

Network Architecture

This chapter provides an overview of the different network communication paths required by a distributed digital identity system built upon the Digidow architecture. It specifies what kind of information is exchanged between different Digidow devices and what level of protection is needed for each interaction. Based on this information, we can decide where the established technologies presented in chapter 2 already provide the necessary functionality and where further research is necessary to achieve the objectives from section 1.2.

3.1 Network Interactions

Before we can start to think about protecting network privacy, we need to identify all the different transactions that are going to take place within a distributed digital identity system. For a distributed digital identity system using the Digidow architecture, we identified five different interactions:

1. The interaction between a sensor and the sensor directory to publish a sensor.
2. The interaction between a PIA and the sensor directory to learn about available sensors.
3. The interaction between an issuing authority and a PIA to provide it with new attributes.
4. Communication between the PIA and its owner.
5. Digidow transactions between PIAs, sensors, and verifiers.

As this thesis only deals with the networking fundamentals, a fully specified protocol is out of scope. Instead, the requirements from a network security/privacy perspective will be discussed for every part of the communication.

3.1.1 Publishing a Sensor

The first interaction is the communication required between the sensor and the sensor directory. This interaction is necessary to either publish a sensor when it is first deployed or to update the information within the sensor directory for a previously deployed sensor. All the information needed by PIAs to use the sensor for Digidow transactions must be exchanged during this interaction. This includes at least the contact information of the sensor, its physical location, its public key, and the type of biometric template it can detect and compare. The sensor directory only has to inform the sensor if a request to either publish or update information was successful or not, it does not provide any further information to the sensor.

3.1.2 Requesting Sensor Information

The next interaction occurs when the PIA wants to learn about available sensors. This requires downloading the information published by sensors from the sensor directory. The information exchanged during this interaction is the same as in section 3.1.1, but a PIA is likely to request and receive information about multiple sensors at once. PIAs will typically store information on sensors locally, so they might also choose to save bandwidth by asking for differential updates to the sensor directory since their last request. This should provide them with all sensors that have either published changes or recently joined the sensor directory.

Note that there might also be sensors that do not wish to be publicly available, like surveillance cameras operated inside private homes. PIAs can support such private sensors by manually injecting sensors into their local copy of data from the sensor directory. This enables them to use such private sensors just like public ones without the need to change other parts of the network protocol.

3.1.3 Issuing Attributes

Issuing authorities need to communicate with PIAs in order to provide them with attributes. Without attributes, a PIA would not be able to prove anything to a verifier, therefore this step is a requirement for successful Digidow transactions. From a functional perspective it would be sufficient to simply transfer the signed attributes from the issuing authority to the PIA, but there are two caveats that need to be considered:

1. The issuing authority has to make sure that attributes are only provided to the correct PIA.
2. The issuing authority has to ensure that no PIA other than the intended destination can present the attributes.

The first part can be addressed by organizational measures to authenticate the PIA that is receiving attributes. One potential approach would be to first send a one-time token to the PIA that is about to receive the attributes via an end-to-end encrypted channel and having it forward this message to its owner. If the owner can present that one-time token within reasonable time, there is a high probability that the issuing authority is talking to the correct PIA. The second issue will most likely have to be addressed on a cryptographic level by tightly coupling the issued attributes with the biometric features of the individual. That should be sufficient to prevent other PIAs from utilizing the same attributes.

3.1.4 Communication with the PIA's Owner

Every digital identity system will sometimes require users to make decisions themselves. The Digidow concept expects those manual decision to be frequent at the beginning because the PIA needs to build up a history of transactions and user decisions. Over time, the PIA should become increasingly capable of making decisions on behalf of its owner without having to ask for the owners' approval every time. Users might have very different requirements in terms of how they want to communicate with their PIA, making it almost impossible to propose one approach that will work for everyone. If a PIA is running on a mobile device like a smartphone, communication is easily possible by just triggering a notification within the local system. PIAs operated remotely would need

another channel of communication. The currently most realistic scenario expects users to still own a smartphone and allow their PIA to communicate with their phone. If the PIA is unsure about an individuals' location, it can contact the smartphone to ask for its current location, and if the PIA is not sure if it should reveal certain attributes to a specific verifier, it can ask for permission before moving forward.

3.1.5 Digidow Transactions

Figure 3.1 shows the flow of a successful Digidow transaction. The main objective behind this proposed flow is to keep the number of messages exchanged between devices as low as possible. One reason for this is tied to the use of onion routing which adds significant latency to every message. For optimal performance, networking has to be designed to use fewer but larger messages to minimize the impact of onion routing on the overall transaction time.

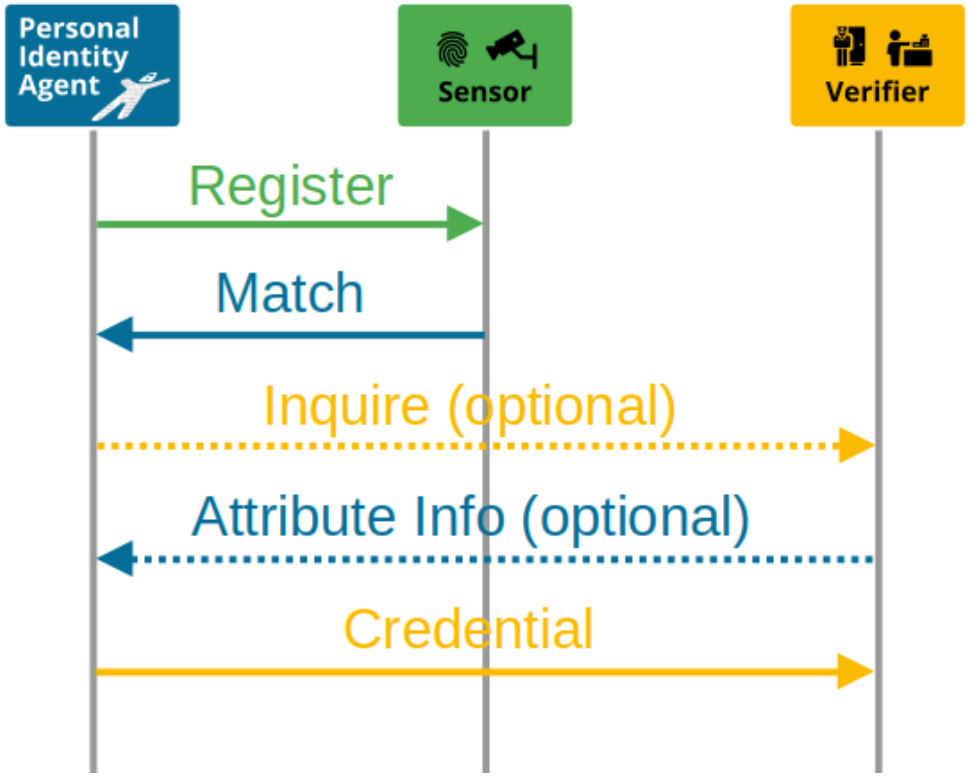

Figure 3.1: The Digidow transaction protocol

The first two messages exchanged between PIA and sensor are inspired by the Publish/Subscribe concept [75] that enables the efficient movement of messages between devices that do not know each other. Within the Digidow architecture, the PIA would infer the current location of its owner based on available data and use the sensor directory to identify sensors that it should subscribe to. It then sends *register* messages to all these sensors. A register message contains some kind of contact information that enables the sensor to reach the PIA if necessary, a piece of biometric information that enables the sensor to identify a potential match and a timeout after which the registration information

should be discarded. Note that it cannot simply include the biometric measurement of the target because biometric information must be considered privacy sensitive, so some form of privacy preserving biometrics [96] will be required. Approaches like fuzzy extractors [35] or cancelable biometrics [109] seem the most promising for the purposes of the Digidow project, but a detailed discussion is not within the scope of this thesis.

After receiving a *register* message, the sensor adds the received biometric template to an internal list that it compares observed biometric measurements against. If this comparison results in a match, the sensor sends a *match* message to the callback provided by the PIA. The *match* message has to provide a proof signed by the sensor that an individual with the detected biometric features has been detected by this sensor at a specific point in time. The sensor should also include information about verifiers that might be interested in this information along with other optional information like its confidence in the biometric match. The key point here is that the sensor only provides the information to the PIA, not to the verifier directly, giving the PIA full control over the decision if the verifier should be presented with any information about its owner.

Once a PIA receives a callback from a sensor, it has to do a bit of internal work before it starts communication with the verifier. First, it has to decide if it believes the information it received. Biometric comparison is always about probabilities which is why sensors should reveal their confidence in a match. PIAs need to account for false-positive results by combining information from multiple sensors with their transaction history to achieve the necessary confidence to continue with the transaction. If the PIA determines the match to be a false positive, it simply logs the message for further reference and takes no further action. Otherwise, it continues with service discovery by trying to identify the verifier that its owner wants it to talk to. In most cases, the sensor will have provided sufficient information to make this step easy and in the rare case where this does not happen and the transaction history is also not helpful, the PIA will contact its owner directly to receive further instructions. Note that there are plausible scenarios where a PIA may either not find a verifier for a *match* message or intentionally decide not to interact with it. For example, a user walking by a bus stop while going to work would be regularly detected by a camera sensor at the bus stop and registering with that sensor would be useful for the PIA because it confirms that its owner is on the way to work, but there is no reason to communicate with the verifier. Sensors in public places that see lots of foot traffic might be intentionally deployed to update the physical location information of the PIA without any intention of ever being used for transactions.

After the PIA is certain that a verifier should be contacted, it has two different options of reaching out. If it has not had any transactions with the verifier recently, it sends an *inquire* message to the verifier to find out which attributes the verifier is interested in. The verifier has to respond with the set of attributes it needs to know about along with the issuing authorities it accepts attributes from. Verifiers that are already known to the PIA because it has had transactions with them in the past, are not asked about their attributes again and the required attributes are retrieved from the PIA's transaction history instead. Until this point the verifier has no information about the PIA, it does not even know if the PIA has a valid proof from a trusted sensor so there might not even be the potential for a transaction at all. The PIA now needs to decide if it is willing to provide the requested information to the verifier. This is an important step because it marks the point where the PIA loses control over what happens with its information after sending it. The verifier might store it, share it, publish it, sell it, or take any other action that could compromise the privacy of the individual owning the PIA. This raises the question of how PIAs should make

the decision between asking for confirmation and deciding on their own. Once again, the task of striking the right balance between protecting privacy and user convenience is explicitly out of scope for this thesis, but research within the Digidow project is actively tackling this challenge. Once the PIA has sufficient reasons to believe that its owner wants to share the requested attributes with the verifier, the PIA creates a cryptographically verifiable presentation of the requested attributes—at this point in time the data format proposed by the W3C for verifiable credentials [115] seems like a good fit—and forward it to the verifier. With the arrival of the presentation at the verifier, the networking part of a Digidow transaction is concluded. The verifier still has to decide if the attributes are valid and then trigger the corresponding action, but this does not require further communication with other devices within the Digidow architecture.

3.2 Service Discovery

Before Digidow transactions as discussed in section 3.1.5 can begin, the three involved parties have to be identified. This process of service discovery happens for every potential Digidow transaction and requires that sensors, PIAs, and verifiers have already been deployed and PIAs have already been issued attributes by the issuing authorities. The purpose of service discovery is to identify the correct triplet of endpoints that will be needed for the current transaction. A sensor to confirm the physical location of an individual, a PIA to provide verifiable attributes about this individual, and a verifier that acts based on the provided attributes. If the correct endpoint cannot be determined reliably, it might also be acceptable for service discovery to produce a small set of options that can be narrowed down later during the verification stage. Until service discovery is certain that it has found the correct endpoints, service discovery is not allowed to reveal any privacy sensitive information about the transaction for which service discovery is currently happening. Otherwise, some Digidow participants might be able to learn about Digidow transactions that they are not involved in. At the same time, the service discovery process is a part of the networking scheme and must adhere to the objectives specified in section 1.2. This puts strong limitations on the time available for service discovery, which in turn causes a trade-off between privacy and user experience.

3.2.1 Initiate Service Discovery

The first step of service discovery must be taken by a device that is aware that service discovery is needed and has enough information to detect at least one of the other two endpoints. After the second endpoint has been found, the third one only needs to be detectable with the shared information of both devices. In order to provide a well founded answer to this question, this section analyzes the information available to every party and then discusses the benefits and drawbacks of it being responsible for the first step of service discovery.

Information available to the sensor

The first action that initiates a Digidow transaction is an individual interacting with a sensor, which enables sensors to tell whether service discovery has to be started or not. Additionally, the sensor has the biometric information of the

detected individual, providing it with a reliable identifier that could be used to search for the PIA of the individual. Lastly, sensors have to be trusted by verifiers in order to conduct a successful Digidow transaction making it very likely that sensors are deployed to facilitate services provided by specific verifiers. By knowing why they were deployed, most sensors are likely to have an idea about which verifiers are involved in a transaction that includes them.

Information available to the PIA

A PIA holds the identity information of an individual as well as the policies that decide which attributes are shared with various verifiers. In addition to that, a PIA can keep a log of previous transactions, effectively providing it with history of what its owner has been doing in the past. Since there is no restriction on where a PIA is operated, a PIA might also be running on a mobile device that the individual carries around, potentially providing the PIA with information about the current location of an individual.

Information available to the verifier

The verifier has a strong trust relationship with the sensor, meaning that a verifier will typically maintain a list of sensors that it trusts. This list could be used to significantly restrict the candidate pool of potential sensors. Since the verifier is not allowed to know anything about the individual that triggered a Digidow transaction. Before that individuals PIA decides to share some attributes, it has a hard time detecting PIAs. Its knowledge about the attributes that must be presented for a successful transaction could still enable it to detect PIAs. If a transaction can only be completed by PIAs that hold certain attributes, the verifier can safely exclude PIAs without these attributes.

3.2.2 Evaluation

In order to determine the optimal approach towards service discovery, all potential variants must be investigated. Figure 3.2 visualizes that there are six different sequences in which service discovery could take place. Note that the process of service discovery can be in three different stages. In stage *A* none of the three devices to be involved in a transaction know about each other. During stage *B* two of the three devices have already found each other, but the third is still missing and finally in stage *C* service discovery has completed when all three devices know about each other. The focus of this section is on deciding how the transition from stage *A* to *B* should be achieved.

Start with the Verifier?

Although a verifier possesses useful information to restrict the number of sensors that could be involved in a transaction, initializing service discovery from the verifier is very hard, because they have no way of knowing when a new transaction is starting. The only way for them to do this, would be to keep constant connections to all their trusted sensors open at all times to learn about new events at the sensor immediately. Unfortunately, even this is not really an option because verifiers are much less trusted than sensors, because they are neither running known code not using a hardware root of trust. This means that

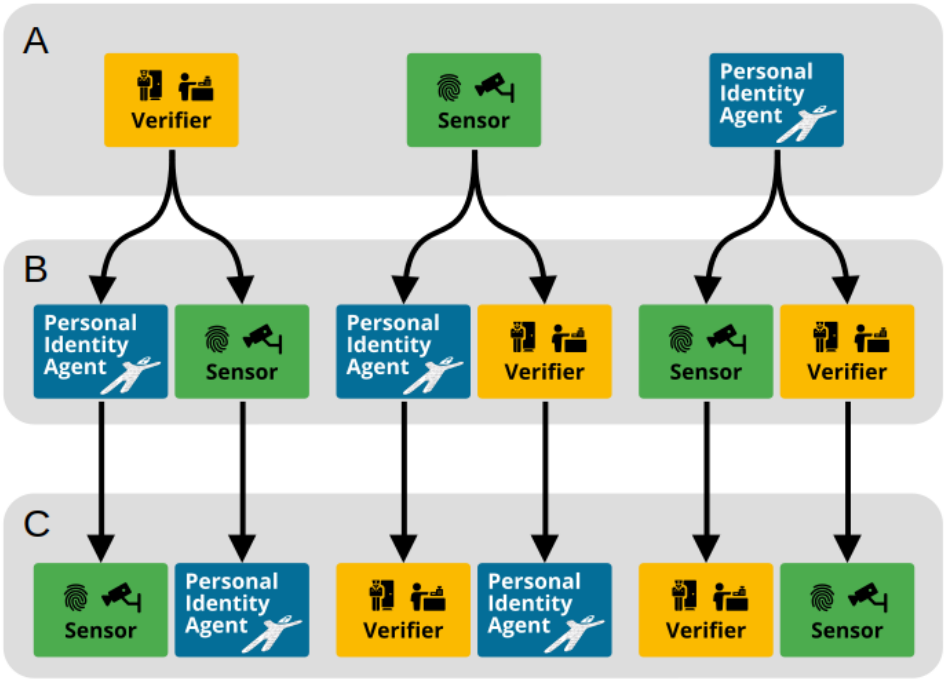

Figure 3.2: All possible service discovery options

the sensor is not allowed to share any information with the verifier before the PIA has confirmed that it wants to execute a transaction with this verifier. This does not apply in the other direction however, so the verifier might be allowed to share information with the sensor that it already trusts to enable the sensor to advance service discovery. One potential scenario where this could work is access control for a private home where the verifier is operated by the issuing authority that issued the permission to open the door in the first place. In this situation, the verifier could provide the sensor with a list of PIAs that would be allowed to unlock the door. The sensor could then establish connections to all candidate PIAs and compare if any of them is responsible for the individual it has detected. If there is no match within the set of candidate PIAs, there is no need to continue service discovery because the verifier will definitely deny the request for every other PIA.

Start with the Sensor?

Using the sensor as a starting point for service discovery is the obvious choice because it knows exactly when service discovery is necessary. Every time a new biometric is detected, the sensor can initiate a service discovery attempt. Some of these attempts must be expected to fail if there are individuals that either do not use a PIA or have a PIA that is currently unavailable. As long as a failing service discovery attempt does not compromise the privacy of individuals, failing service discovery attempts are most likely acceptable, especially if they do not constitute a distributed denial-of-service (*DDos*) vulnerability because they are triggered by observing an individual in the physical world. There are two different ways how a sensor could approach the task of service discovery: It could use the biometric information it just collected and try to find the PIA that

feels responsible for it or it could try to detect the verifier that the individual wants to prove its attributes to.

Identifying the verifier is probably the easier option, because every sensor knows who it is being operated by, which often hints at the verifier that is needed for the transaction. Since the verifier can also detect the sensor, this raises the question whether sensor or verifier are better suited to initialize service discovery if they are both finding each other. Apart from the fact that sensors know better when service discovery is needed, there is a second argument for having the sensor being responsible for the first step. Most sensors are operated for a specific purpose and this purpose is tied to only one or at least a very small number of verifiers, while verifiers are more likely to trust a lot of sensors. Using public transport as an example, checking the digital tickets of all passengers requires an operator to have sensors either at the access to all stops or inside all vehicles. Both options will easily result in hundreds of sensors deployed that all refer back to the same verifier. Constructing an example where a sensor would deal with a similarly sized set of potential verifiers is actually challenging. Checking in at larger airports might mean that airport sensors are being used by one out of hundreds of different flight operators, but sensors at specific gates would only have to support a handful of verifiers at the same time. During the course of this research, not a single realistic scenario could be identified where service discovery started by the verifier would have an advantage over service discovery initiated by the sensor, leading to the conclusion that the sensor is better suited to initiate service discovery than the verifier.

A vastly different situation presents itself if the sensor tries to find the PIA responsible for a specific set of biometric features. Without any way to filter the number of potential PIAs first, the sensor would have to search a dataset of potentially billions of PIAs in order to find the correct one. While there is no system in place that regularly compares one biometric against billions of others, systems like IAFIS [67] have several hundred million biometric templates on file and law enforcement regularly searches the entire database for matches with biometric evidence collected at crime scenes. These searches have been heavily optimized and take between 1 and 2 seconds each [119] which is already longer than service discovery should take in total. Even more serious is the problem that a distributed digital identity system cannot use a centralized biometric database that constantly gets queried by sensors, because that would give too much power to the entity controlling this database. An attempt could be made to mitigate this issue by decentralizing the biometric database, but that would most likely have a significant negative impact on the times needed to find the matching PIA for a biometric measurement. Even if the distributed database would provide the same performance, there would be a public repository that matches biometric features to the onion address of a PIA, which would seriously compromise the privacy of transactions within the Digidow network. This leads to the ultimate conclusion that it would not be desirable for sensors try and lookup PIAs purely based on biometric information.

Start with the PIA?

The history of transactions known to the PIA provides it with historic location information about the individual, allowing it to predict certain actions with reasonable certainty. For example, an individual leaving their house at 8:00 AM on a weekday is most likely going to work. If that trip usually happens by bike, the next sensor that will encounter the individual is located at the employer. If the trip uses public transport, the next sensor to talk to would be the one at the

nearest public transport station. Pulling in relevant circumstances like the current weather could make these predictions even more accurate. With those predictions, a PIA could connect to sensors that are likely to observe their owner, if they had a way of knowing the physical location of available Digidow sensors as well as a way to contact them. This would require a public dictionary of available Digidow sensors that includes their location and their contact information. Such a list would have to deal with similar challenges as the biometric database needed by the sensor to discover PIAs. It would have to be distributed to prevent a single entity from abusing their control over the directory to hide sensors or track how often sensors are being used. Even then, the contact information for sensors compromises unlinkability and makes it easy for adversaries to launch attacks against sensors. To recover service discovery in situations where the PIA does not predict the movement of an individual correctly, individuals could use mobile devices like smartphones to inform their PIA about their current location, which should be enough for the PIA to connect to the correct sensors.

An almost identical strategy could also be applied to discover the verifier involved in a transaction. If a PIA can determine the physical location of an individual, it could use that information to reason about potential verifiers that its owner might interact with in the near future. For example, if an individual is close to a subway station, the PIA could get in contact with the verifier of the subway operator. This would require a public directory of verifiers that ties them to physical locations where they provide services, similarly to how the detection of sensors would require a public record of their location. Once the correct verifier has been determined, the detection of the correct sensor would most likely be easier because the verifier could just provide a limited set of sensors that it trusts. This approach would be more efficient than discovering sensors because it seems reasonable to expect there to be more sensors than verifiers for a public digital identity system, but it would also put verifiers in a more powerful position than necessary. Considering that PIAs are making educated guesses rather than knowing for certain what an individual will do, there will be a significant number of false positives, especially for verifiers like public transport providers that service large areas, as they are likely to be consistently contacted by everyone moving around their service area. Even if verifiers were unable to link those requests to certain individuals, verifiers would still obtain information about public movement patterns that would not be exposed if PIAs identified their sensors directly. Ultimately, both variants of service discovery triggered by the PIA appear to be viable, with detecting the sensor first being more privacy preserving, while detecting the verifier first would most likely reduce the time needed for service discovery. Based on this consideration, the first implementation of service discovery will try to detect the sensors directly and only if this approach is not fast enough, discovering the verifier instead will be evaluated.

3.2.3 Finalizing Service Discovery

Figure 3.3 shows that analyzing potential candidates for initiating service discovery has already ruled out 3 out of 6 possible options. And one of those options—the option to have the PIA discover the verifier first—has already been found to be less desirable than its alternatives. This leaves two competing approaches that need to be evaluated in order to reach a final decision:

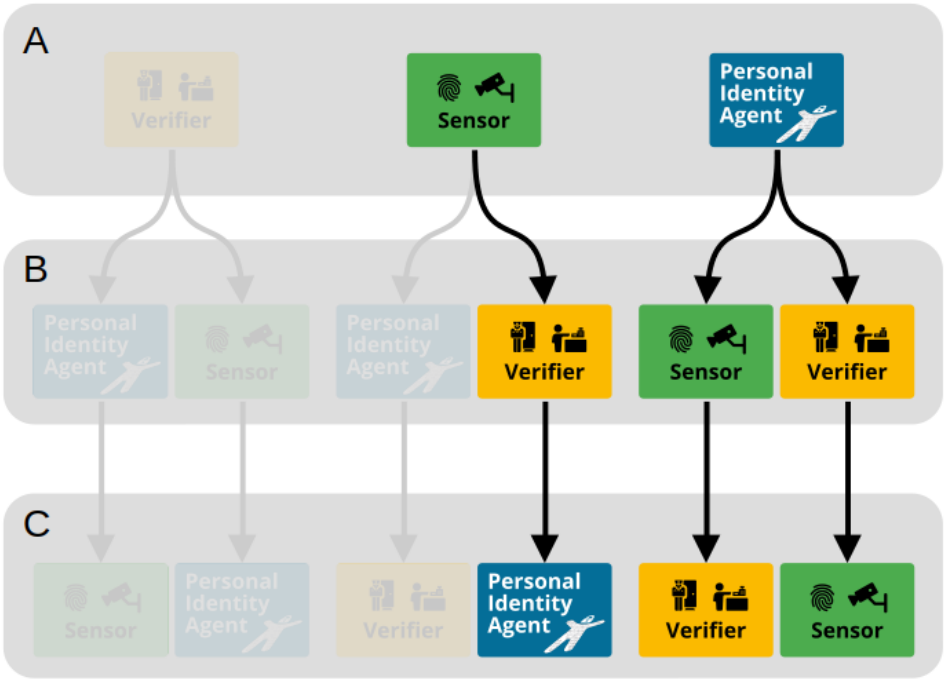

Figure 3.3: Remaining possible service discovery options

Information available to sensor and verifier

Obviously, a cooperation of sensors and verifiers has access to all the information they had individually (see section 3.2.1), so the key aspect here is the additional information they can produce by cooperating. Unfortunately, this cooperation is hampered by the fact that a sensor must not reveal the biometric information it collected to the verifier, since the permission to do this can only be given by the PIA after it has been discovered. Apart from the biometric information, the sensor does not have any information about the individual. The other directions would be less of an issue, so if the verifier has any information about the PIA it could share that with the sensor. As discussed in section 3.2.2, the verifier can limit the number of candidate PIAs based on the attributes that it would accept from the PIA if it could be found, which would enable the sensor to only compare its biometric against a limited set of PIAs.

Information available to PIA and Sensor

The cooperation between PIA and sensor can take a relatively simple approach to discover the correct verifier for a transaction. In most situations, the sensor will know the only verifier that trusts it and therefore be able to directly provide the answer. However, in the unlikely event that the sensor does not know the correct verifier to communicate with, the PIA can still check its own transaction history to see if this sensor or similar sensors were used with specific verifiers in the past. If the sensor is unsure between multiple verifiers, the PIA could check if it holds any attributes that would indicate a specific verifier. An example for such a situation would be an event where three different vendors were given permission to sell tickets. The sensors at the entrance would

be aware that every ticket has to be confirmed by one of the three verifiers, but they would have no idea where a specific individual purchased their ticket, so they could only provide a list of several candidate verifiers to the PIA, which would be able to select the correct verifier based on the ticket that it has stored.

Use Sensor and Verifier

As discussed in the previous sections, the information that can be shared between a sensor and a verifier reveals very little about the individual interacting with the sensor. The only easy scenario would be a verifier that knows the small group of PIAs that could potentially provide valid attributes. Scenarios like access control for personal office space could be handled this way, but even this approach comes with a downside. If PIAs need to provide their contact information to verifiers, they have no way of knowing how the contact information could be abused later on. Contact information like onion addresses may be anonymous, but they are still linkable so a collaboration of malicious verifiers would be likely to find out if they have a shared subset of users. Even if that issue could be solved, there are still scenarios where this approach towards service discovery is not feasible. Verifiers for public transport in cities for example are very likely to handle up to several million users every day. Such verifiers would end up having to go through a huge list of PIAs and doing a biometric comparison with each of them, an approach that does not scale and would most likely take way longer than acceptable. Lastly, there are plausible scenarios where verifiers might have no idea about the PIA because there was no enrollment step that would have told the verifier about a new candidate PIA. Age verification in front of clubs is one example for such a scenario. A club would most likely accept any government issued attributes that confirms the individual's age to be above 18, setting the pool of potential PIAs to almost all running PIAs. At this point the issue discussed in section 3.2.2 comes into play again. If the verifier cannot provide the contact information of a few PIAs that the sensor should contact, the sensor only has the biometric template to work with and—as discussed in section 3.2.1—searching PIAs in a global database indexed by biometric features is not feasible.

The only way to resolve these problems would be to introduce auxiliary information that helps sensor and verifier to discover the PIA. This could be as simple as a QR code scanner included within the sensor that can be used to give the sensor the contact information of the PIA, if it cannot detect the PIA automatically. To keep user interaction down to a minimum, sensors or verifiers would have to cache new contact information for PIAs that successfully completed a transaction. Otherwise, users would have to keep providing their PIA's contact information which would most likely not be accepted by users. Even with this auxiliary information, service discovery would still have a scalability problem if there are too many PIAs that need to be contacted individually. This could be solved by giving up on caching for larger verifiers and just requiring users to always present their PIAs contact information, but in this case identifying individuals directly based on a token they carry would be the more efficient solution.

Use PIA and Sensor

PIA and sensor can always discover the verifier with the same certainty as the sensor can on its own. Even when the sensor is uncertain because there are several candidate verifiers, the PIA can either decide based on its transaction

history and stored attributes, or it can contact all candidate verifiers and ask them for the attributes they require and the service they provide. This approach would support all the different scenarios that required special attention before without removing support for any other type of scenarios. Consequently, a service discovery process where the PIA identifies potential sensors first and then discovers verifiers based on information from the sensors should be best suited to support all potential use cases of a distributed digital identity system.

3.3 The Sensor Directory

As discussed in section 3.2.1, PIAs require a public directory of available Digidow sensors along with their contact information and their location to identify and connect to sensors that their owner is likely to interact with. The research conducted for this thesis resulted in the inclusion of a *sensor directory* within the Digidow architecture in Figure 2.1. While the implementation details for the sensor directory are not part of this thesis (as they are still part of active research), the requirements it is expected to meet will be presented here:

3.3.1 Functional Requirements

- The sensor directory should maintain a list of available Digidow sensors comprised of the geographical location, type (which kind of biometric is it recording), identity key and contact information. Every entry within the sensor directory should be signed with the private key that matches the public key listed in the entry to prevent unauthorized modifications. Optional fields for the operator of the sensor, the purpose of the sensor, or the attributes that its verifier will require should also be available.
- Digidow sensors should be able to register within the service directory, by either creating a new entry or updating their old entry.
- PIAs should be able to query for all Digidow sensors available in a specified area or for all sensors within the proximity of a specified location.
- PIAs should be able to request all updates made to the sensor directory without having to download the entire list of sensors.
- PIAs need to be able to request a checksum over the entire database that can be used to validate the overall state of the database.

3.3.2 Privacy Requirements

- No single entity must be able to monitor or manipulate requests made against the sensor directory.
- Sensors must never be removed from the database, they can only be updated to be discontinued.
- The sensor directory must be able to prove to PIAs that it is operating as intended.
- The sensor directory has to respond to requests from PIAs quickly enough to not negatively impact the performance targets set in section 1.2.

3.3.3 Potential Approaches

Some readers may have already started thinking about the Blockchain [94] when the requirement for a distributed public directory was introduced. From a functional perspective, a blockchain could indeed be used to store the information about available sensors, but it does come with the classic challenges. A proof-of-work blockchain wastes a lot of resources that could definitely be spent more efficiently on other tasks and it would have to find a way to reimburse the users who contribute their resources. A proof-of-stake based blockchain might be an acceptable solution, but would still introduce the challenge that these stakeholders would suddenly be able to prevent sensors from registering within the sensor directory.

Another established solution for sharing this kind of information is the approach taken by PGP keyservers [113] that provides far less structure but gives PIAs more choice in their selection of information sources. Since the information from the sensors could be independently verified by every client, there is no need to go through a proof-of-stake procedure before including a new sensor in the list. Obviously, the time until a new sensor has propagated to all keyservers might be an issue and so is the question who would operate them. If there are not enough of them, individual key servers might again be able to infer information about individuals location based on the requests made by their PIAs.

Lastly, the approach taken by the Tor project to publish service descriptors—the hidden service directory—could also serve as an example for how the sensor directory could be designed. It definitely serves as an example how critical information about individuals from such a shared database [11, 102] can be unintentionally leaked. Unfortunately, many of the strategies of the HSDir do not apply to the sensor directory, because sensors do not have knowledge of any keys that the sensor directory would not be aware of and data would be retrieved in chunks which would make it more complicated to randomly spread out information across multiple participants. The process of evaluating the various possible implementations of the sensor directory is a research question of its own that is currently pursued by other researchers within the Digidow project and therefore no subject of further discussion in this thesis.

3.4 Threat Model

3.4.1 Threats

In every digital identity system there are two types of participants with legitimate interests. Users (PIAs) who want to access services and verifiers who grant access to services. Both of them can suffer from privacy violations and therefore require protection of their data and legitimate interests.

Threats against users

Users—or more specifically their PIAs—could be targeted in several ways. The obvious network attack is a passive adversary monitoring their communication to learn the contents and patterns of their conversations. Even if the contents of the conversation cannot be accessed, information about when and how often a PIA communicates with sensors, verifiers or issuing authorities compromises the user's privacy. Similarly, the absence of communication can also

reveal privacy sensitive information about a user, by telling an attacker that a user has not used any of its digital identities during a specific period of time. Even communication not directly related to the digital identity system, like resolving hostnames via DNS, requires connections to a third party that leak private information about a user.

Unfortunately, traffic monitoring is not the only way in which PIAs could be targeted. Active attackers could conduct man-in-the-middle attacks to modify the communication between entities and trick devices into revealing more information than they should or just outright sabotaging the availability of the digital identity system entirely. As a final resort, adversaries also have the option of actively dropping network traffic from selected PIAs to render their owners unable to use their digital identities. The same could also be achieved by standard distributed denial-of-service attacks, which most PIAs are unlikely to withstand.

Threats against verifiers

Verifiers are mostly expected to be operated by organizations rather than individuals, but the threats they face are still similar, although their priority shifts as availability becomes more important than privacy. If the business of a company depends on the availability of verifiers so customers can access its services, attacks that prevent network access for clients are more of an issue for them, than it is for the clients. There are also legitimate confidentiality interests that apply to organizations, mostly in terms of how many customers they have, at which point in time the customers use its service, and who those customers are. The verifier itself receives all the necessary information via the attributes provided by the PIA, so there should be no other information leaking out that competitors could use to learn more about a specific business.

3.5 Attackers

This section lists the various types of attackers that could try to attack the network of a distributed digital identity system. They are grouped by their capabilities, not by their intentions because most attacker types could potentially target either PIAs or verifiers.

- *Perfect global passive adversaries* are entities that can observe the entire global network traffic at a given point in time. There are no public indications that such entities currently exists in the real world.
- *Global passive adversaries* are attackers capable of passively observing a significant amount but not all global network traffic. Some intelligence agencies have been shown to possess this capability [88].
- *Local passive adversaries* are attackers with the possibility to monitor network traffic at one specific location. There exists a wide range within this category from large entities like internet service providers or operators of internet exchange points down to operators of public WLANs or attackers controlling the router in a private home. Typically, global passive adversaries obtain their information by forcing multiple local adversaries to hand over their data.
- *Global active adversaries* are capable of modifying a significant fraction of the global network traffic and have the resources to deploy a significant

share of network devices to add their own traffic. This kind of adversary might deploy their own malicious sensors, PIAs, verifiers or Tor relays to achieve its goals. China's great firewall [37] is one example for such an adversary.

- *Local active adversaries* can manipulate traffic at one specific location or deploy a few malicious devices within a network to obtain more information.

3.5.1 Countermeasures

Fortunately, many of the threats listed above can be tackled with established strategies. Preventing eavesdropping, tampering, and message forgery can be achieved by adopting standard cryptographic protocols like TLS [110] (as long as certificate checks are performed correctly). This prevents active attackers from modifying messages and passive adversaries are left unable to read the contents of the messages exchanged between communication partners. This still leaves the meta-information of who is communicating with whom unprotected, so the network anonymity tools from section 2.2 also need to be utilized. As discussed in section 2.3.2, the Tor network was identified to be the best currently available tool for digital identity systems, although it does not protect against *perfect passive global adversaries*. Protecting the anonymity of both communication partners is best achieved by communicating via onion services, which also adds a default end-to-end encryption to all communication, removing the need for an additional TLS layer. Onion services are also beneficial in another way as they remove the dependence on external services like DNS that would negatively impact privacy.

A significant part of our privacy objectives can be achieved by using the Tor network and onion services, but there are some areas where the Tor project's primary goal of network anonymity is actually not sufficient to ensure network privacy in a distributed system. Consider for example passive adversaries that have the capability to monitor a part of Tor's hidden service directory. Without any further access to the network traffic of a sensor, they could find out when and how often a specific sensor is being contacted. For a sensor that is used to open an office door, this information would leak information about the working hours of the individual working behind that door. To protect against such attacks by passive adversaries, the networking architecture for a digital identity system should aim for a new property that will be defined in this thesis as *network unlinkability*.

3.5.2 Network Unlinkability

RFC 6973 [22] defines unlinkability as the inability of an observer to tell if two items of interest (IOI) are related or not. If IOI's have at least one unique identifying attribute, unlinkability can only be achieved within specified sets of information. In the context of network unlinkability, protocol information like IP addresses or TCP ports are standard examples of IOIs. The amount of traffic exchanged via a TCP session or the times when data is sent are further examples for IOIs that enable an observer to reason about the activity of a client. The adaptation of onion routing does not prevent an observer from measuring any of these values because onion routing still uses TCP connections between the various circuits, but it reduces the amount of information they carry. Since a Tor client will typically only use two different *guard relays*, an observer monitoring outgoing traffic will only see active TCP connections between the client

and its guard, which tells the observer that communication via the Tor network is taking place. It is impossible to determine who is being communicated with or how many different communication partners there are, but the times during which communications happen as well as the amount of data exchanged remains visible to an observer. Since Tor uses padding to make most of its cells have the same size, the only IOI for which a passive attacker can identify relations easily is the timing of transmitted cells. More active attackers have other options as well, most notably via the hidden service directory. Joining the HSDIR with a few relays is easy for attackers with moderate resources, and allows them to see the uploads and downloads of their fraction of the distributed hash table. Within this fraction, they can easily find out if upload and download requests are made for the same service descriptor (they need to know that in order to respond with the right answer) enabling them to track the usage of some onion services every day.

Network Unlinkability vs. Network Anonymity

An important observation to make at this point is that networks like Tor were designed with network anonymity in mind, meaning that adversaries should be unable to discover the network location of Tor users. This does provide a measure of unlinkability, because some IOIs need to be removed to conceal the clients network location. In other regards, like the hidden service directory, the difference shows quite openly. Tor protects the anonymity of the operator and visitors of an onion service, but it does not protect the information about when and how often an onion service is being connected to. Most likely, Tor had to decide between performance and more unlinkability in this case and decided to accept a degree of linkability in return for faster translation of onion addresses into service descriptors. Another example for network linkability within the Tor network are the guard relays used by onion services. These guard relays know about outgoing circuits established by the Tor client, and onion services have a very peculiar pattern when it comes to circuit creation. There are usually three long-lived circuits that are kept open for weeks or even months that are only ever used to receive incoming cells without any outgoing cells, and most incoming cells cause the creation of new circuits. Every guard relay can use the IOIs of traffic sent via circuit and time of circuit created to detect if a client is operating an onion service (the regularly occurring 16 uploads to the hidden service directory might be a hint as well). None of this weakens the anonymity of the onion service (the guard knows its network location anyway), but it demonstrates that providing anonymity does not provide unlinkability. Ultimately, this discussion circles back to the difference between onion routing and mix networks. As long as onion routing forwards traffic in real time, the timing of messages becomes an IOI that can be used to link traffic, and even if it does not compromise the anonymity of a Tor client, it might still leak privacy sensitive information about users of digital identity system. Every interaction within the network architecture must be designed with this limitation in mind to minimize the privacy impact of time based correlation attacks on the privacy of Digidow's users.

3.5.3 Acceptable Risks

As a final part of the threat model, it is important to acknowledge risks that are inherent to every identity system and should not be overstated. When evaluating both security and privacy risks introduced by a distributed digital identity system, the key objective should be to ensure that there are no new security or

privacy issues that are either easier to exploit than before or work on a larger scale than before. For example, we do know that onion routing is vulnerable to timing analysis, so there is a risk that adversaries might deanonymize communication routed via the Tor network. This could tell an attacker that one specific PIA communicated with a sensor or a verifier for example. While this is considered a risk under this model, the same information could also be obtained by simply following the person in the physical world and seeing it successfully interact with a sensor. This approach taken by physical adversaries is much more reliable—as the attacker can select the target in advance—and almost impossible to avoid. Fortunately, it does not scale at all, so adversaries are only able to monitor a very small set of individuals. The privacy protections provided by the Tor network (not being able to selectively or reliably deanonymize users) ensure that surveillance by monitoring network traffic is less efficient and reliable than physical surveillance. We conclude that there is no need for our proposed Digidow architecture to be free of all risks, it is sufficient if the risks included are so small compared to the risks inherent to identity systems that no reasonable attacker would choose to exploit them. Such risks are considered as acceptable risks for the network architecture.

The usage of onion routing over mixnets is a good example for such an acceptable risk. Yes, onion routing will allow attackers to sometimes deanonymize some interactions, but the vast majority of communication will be protected. Adversaries with the capability to launch large scale attacks on the Tor network would focus their attention on other mechanisms like physical surveillance or forcing verifiers and issuing authorities to hand over data. As long as onion routing prevents both large-scale and targeted deanonymization of communication, it reduces the risk of passive adversaries to an acceptable level.

3.6 Securing Network Interactions

3.6.1 Publishing a Sensor

The interaction between a sensor and the sensor directory requires very little protection. Information exchanged between these parties is intended to be published, so there is no need for confidentiality. Even if the publication fails, the sensor already intended this information to be public, so there is no reason to keep the content confidential. Furthermore, the sensor directory does not receive any privacy sensitive information during this interaction, it only learns information about a sensor which has no right to privacy. Significantly more important than confidentiality is the integrity of the published information. To ensure that a sensor provides meaningful information, their published information must carry a signature that can be verified with the public key they are publishing. Most likely, the published contact information should also be tested before the sensor is accepted within the sensor directory, but this is not relevant for the network protocol, and therefore remains future work for the research on the implementation of the sensor directory. From a network anonymity perspective, sensors have no reason to hide that they are communicating with the sensor directory. Their existence, location, and identity is revealed by the sensor directory already, and communication with the sensor directory is part of their normal operation. Similarly, the sensor directory has no need for network anonymity because its operators need to be known publicly anyway to ensure that users can trust them to not abuse their position within the Digidow infrastructure.

There is one argument for publishing sensor information anonymously and that is that it makes it harder for passive adversaries to tell the difference between a sensor publishing new data and a sensor contacting a PIA. However, since publishing to a sensor directory does not happen very frequently and can be tracked via the public service directory, this argument has little weight. This leads to the conclusion that no specific protections are necessary for this interaction.

3.6.2 Requesting Sensor Information

Downloading information from the sensor directory is more sensitive than uploading because it might reveal information about the activity of a user. For example, if a PIA asks the sensor directory for sensors located at an airport, then the sensor directory can assume that the owner of the PIA is most likely on their way to the airport. Several mitigations must be combined to prevent the sensor directory from leaking this kind of privacy sensitive information. First, PIAs should maintain a local copy of the list of sensors at locations that their owner visits regularly and periodically synchronize their internal list with the up-to-date information on the sensor directory. This periodical update must be sufficiently randomized so that there is no time based correlation between the requests sent to the sensor directory by a PIA and the individuals physical location and movement. Obviously, trips that take an individual around half the globe would still force the PIA to respond with specific requests to the sensor directory, so further protections are needed. Different requests sent to the sensor directory must be unlinkable for both passive observers and the sensor directory itself to prevent patterns from being observed. To achieve this, PIAs are only allowed to establish connections to the sensor directory via the Tor network. With network location and request timing being handled, there remains one other IOI that could be used to link multiple requests from the same PIA, namely the area in which it is interested. If a PIA always asks for sensors in a 5 mile radius around the current position of its owner, requests can be linked based on the assumption that the owner is not very likely to move while asleep which ultimately reveals the home location of the PIAs owner. To prevent this, PIAs could ask for random positions in the proximity of their owner and adapt the search radius to still learn about all the relevant sensors. While this works well in densely populated areas with many PIAs making overlapping requests, sparsely populated areas would not benefit from this approach if all requests in a larger area are linkable because there is only one individual living there. The safer way to synchronize with the sensor directory is therefore to group sensors by established geographic areas like for example postal codes. Asking for all sensors in Vienna for example, would be a request that millions of PIAs could plausibly make, and if the requests come via Tor, a malicious sensor directory is left without any way to find out if two requests are related or not.

One question that will arise several times during the discussion of the networking scheme is the use of Tor onion services for connections that only require privacy for the client. When retrieving information from the sensor directory, the sensor directory might—as argued in section 3.6.1—have no need for network anonymity so the connection could just be established via a regular Tor circuit, which should be more efficient considering that onion service connections are more complex to set up. On the other hand, the Tor network is currently mostly limited by the bandwidth of exit nodes. Using Tor onion services instead of regular Tor circuits might therefore be more efficient, especially with Tor's *HiddenServiceSingleHopMode* that allows an onion service to willingly give up its own network anonymity in return for better connection speeds. For the

purpose of this thesis, we are working with the assumption that onion service connections are the more efficient choice, but there is no solid research available yet to back this assumption. Based on this assumption, we expect PIAs to connect to the sensor directory via Tor onion services.

3.6.3 Issuing Attributes

The attributes issued to a PIA are private and sensitive information, so this interaction must protect integrity and confidentiality of the exchanged data. Network privacy is also an issue because information on which PIA received attributes from which issuing authorities is privacy sensitive. If you receive attributes from the Austrian state printing house for example, you are most likely to be an Austrian citizen. To account for this requirement, a PIA should only accept issued attributes via a Tor onion service, allowing issuing authorities to make incoming connections. This conceals not only the network location of the PIA, but also the network location of the issuing authority. While large official issuing authorities are likely to operate in public, there is no need for issuing authorities to reveal their contact information or network location. Some issuing authorities may have good reasons to conceal when and how often they issue attributes. As an example, consider an organization that offers consulting to woman considering abortion. In some countries, such a consultation is mandatory before an abortion can be conducted legally, so the consultants have to issue an attribute that a woman received such a consultation. Obviously, the privacy of the women would be significantly compromised if outside observers could monitor when a woman is receiving this attribute.

3.6.4 Communication with the PIA's Owner

The communication between a PIA and its owner requires the highest level of security. It contains information about the owner's physical location or attributes that the PIA wants to share with verifiers, both very privacy sensitive pieces of information. Additionally, integrity or authentication failures would enable attackers to fully impersonate the owner. Aside from confidentiality and integrity, network privacy is again an issue. Communication between a PIA and its owner's smartphone takes place at specific occasions, so being able to link these connections together constitutes another privacy risk. By having conversations between the PIA and the owner's smartphone take place via Tor onion services, this risk can be reduced. However, this communication is very likely to remain linkable for certain attackers even if it takes place via Tor onion services. Mobile network providers can track when messages are sent to a smartphone, even if the messages comes from within the Tor network, and the network traffic of the owner who operates the PIA is again visible to the internet service provider. Since both of those providers have contracts with the owner that contain linkable information, they can quite easily use timestamps to identify when and how often a PIA communicates with the smartphone of its owner. Delegating operation of a PIA to a third party does not resolve this issue, but it adds one more entity that has conspire to compromise the owners privacy. The best way to mitigate this issue for now, is to operate the PIA directly on the smartphone itself, but this comes with significant other limitations in regard to power and available performance. This is one of the reasons why the long-term goal of the Digidow project is a state where user interaction is no longer necessary because the PIAs can resolve all issues on their own. Once this goal is achieved, this privacy issue will be solved.

3.6.5 Digidow Transactions

Every Digidow transaction consists of multiple interactions that need to be secured. The first communication happens between the PIA and the sensor when it sends the *register* message. Both passive adversaries and the sensor itself should not be able to find out if two registration requests were made by the same PIA, i.e. registration requests should be unlinkable. Otherwise, it would be possible to both find out who a PIA belongs to and what an individual is doing in the physical world, by collecting and analyzing *register* requests. As a first step, network anonymity can be achieved by onion routing communication via the Tor network. Technically, the sensor itself does not require network anonymity but due to the considerations presented in section 3.6.2, it is most efficient to use onion services for this kind of connection. This effectively decides that the contact information published in the sensor directory has to be an onion address (although the possibility of publishing onion service descriptors instead is investigated in section 7.2). Unfortunately, this is not sufficient to ensure unlinkability because the information included within the request could also serve as IOIs like the callback information or the biometric comparison data. While unlinkable biometric data is not within the scope of this thesis, unlinkable network callback information definitely is. Simply using a static onion address as callback information is infeasible because it breaks unlinkability. Thankfully, there is no limitation on the number of onion services that a single Tor client can operate in parallel, so PIAs can just create a new onion service for every *register* request. This approach comes with significant performance and scalability costs that will require further optimization.

After a successful registration, the connection between the PIA and the sensor is closed. This is a strong contrast to other established notification systems like Google's Firebase Cloud architecture [44] that expects devices to keep a long lived connection open to receive responses. Unfortunately, this is not feasible in a distributed setup because a PIA will have to register at lots of sensors that will never have a reason to respond to them with a *match* message. Keeping all of these connections open would be wasteful and put unnecessary burden on the Tor network. It should be acknowledged at this point that the chosen alternative of creating onion services also requires at least one introduction circuit per registration, so the number of connections is not reduced as much as one might initially assume. However, an active onion service connection requires two active circuits instead of just one, so the Pub/Sub approach still halves the number of required circuits and Tor can optimize introduction circuits for their purpose while the circuits for onion service communication have to be general purpose circuits. Another argument against keeping connections open can be made by considering the perspective of passive adversaries. If they monitor long-lived connections, they can easily link *register* requests with their *match* responses, and they can even track which registrations resulted in a match. By having more short-lived connections that all transmit a very similar amount of information, attackers monitoring the network have to rely solely on timing to link various messages. These reasons lead us to believe that a Pub/Sub based approach with multiple messages is the best option for this interaction.

Since the *match* response has to be sent to the callback destination included in the *register* request, there is little left to discuss. This message is sent to a Tor onion service, which provides anonymity to the PIA and the client as well as integrity and confidentiality protection by using end-to-end encryption and digital signatures. Unfortunately, there is no functionality like an inverted *HiddenServiceSingleHopMode* that allows Tor clients to give up their own anonymity in order to improve their performance when communicating with Tor onion services. This feature (*Tor2WebMode*) did exist in the past and was

removed because it was incompatible with various brute-force protections introduced by the Tor network [46]. This means that onion service connections always ensure the network anonymity of the client, even if the client does not need it.

Similarly to the communication between PIA and sensor, the communication between PIA and verifier usually requires one-sided network anonymity. Most verifiers have to make their contact information public to enable PIAs to contact them and information on when and how even they are being used can be obtained by monitoring the sensors trusted by the verifier. However, the verifier is not allowed to learn anything about the identity of the PIA before the attributes are transmitted. Most notably, the verifier must not be able to distinguish various *inquire* requests to prevent it from requesting different attributes from different individuals. Since the contents of the *inquire* message are static, it is sufficient to communicate via the Tor network to meet this requirement. This results once again in the debate between onion services vs. regular Tor connections to a public service, and—as argued in section 3.6.2—the best solution is for verifiers to operate an onion service in *HiddenServiceSingleHopMode* that can be contacted by the PIA.

3.6.6 Unresolved Threats

Aside from actively communicating devices, there remain several other targets for potential attackers. The most typical target would traditionally be the owner of a PIA targeted via *social engineering*. One could for example set up a sensor close to a public transport station and have it recommend a verifier with a dangerously similar name to that of the public transport company. This could trick many users into sharing their attributes with them because they believe they are only sharing it with a trusted public transport company. Similar tricks could be applied to get a PIA to reveal more attributes than it should to a verifier. On a more technical level, Tor onion services depend on several other Tor relays that might be operated by malicious entities. Since sensors have to publish their onion addresses in the sensor directory, they are publicly available to attackers. By translating the onion address to a service descriptor, an attacker operating a large number of Tor relays can find out if it acts as introduction point for any sensors. If yes, the attacker can either count the number of *INTRODUCE1* cells received for the onion service or it can decide to restrict access to the sensor by responding with *INTRO ACK* cells without actually forwarding the introduction request.

Finally, the anonymous nature of the Tor network makes onion services especially vulnerable to DDoS attacks. The general strategy of dropping traffic early is impossible due to the end-to-end encryption and the effort of creating an onion service connection is significant because a new circuit must be created for every single one of them. The Tor project is aware of this issue [5] and has tried to mitigate it by enabling onion services to set optional limits on the number of introduction requests that the introduction point is allowed to forward. Unfortunately, this approach drops legitimate and illegitimate introduction requests equally, so it constitutes only a very limited solution. There are proposals to address this on a general level by adding a proof-of-work based challenge to introduction requests [70] if an onion service experiences unusual load, but this has not been implemented yet. In our network architecture, sensors are likely to experience the most denial-of-service attacks. Their addresses are publicly available, they are easy to contact and easy to overwhelm. Even if they can handle the attack on a pure network level, a lot of registrations increase the number of comparisons a sensor has to do for every biomet-

ric measurement and it is very likely that a distributed denial-of-service attack can quite easily prevent a sensor from identifying legitimate users trying to interact with it. Hopefully, the Tor project will put forward DDoS mitigations that are powerful and flexible enough to fulfill the requirements of a distributed digital identity system like Digidow.

Summary and Next Steps

The presented networking scheme provides several significant benefits. It reduces the number of mandatory messages exchanged between devices down to three, with the first message being sent ahead of time so the user only has to deal with the latency of two messages until the verifier makes a decision. This should keep the overhead introduced by onion routing within the Tor network down to a minimum. It also works well to protect the privacy of PIAs because they do not show up in any public directories (and even if they would, they only accept incoming connections when receiving attributes or listening for responses to their registrations) making it almost impossible for active adversaries to connect to them directly. They would have to operate a malicious sensor that passes the hardware root-of-trust check by the PIA to get valid onion addresses of PIAs and even then the attack surface would be limited because there is only one message the PIA expects to receive from the sensor. If a PIA receives any other message, it can just disable the onion service, preventing the attackers from communicating with it any further. Unfortunately, the scheme also leaves several challenges open whose resolution will be the main content of the remaining thesis.

Chapter 4 will discuss the privacy challenges tied to the hidden service directory. Which kind of information does it reveal, could attackers realistically obtain that information, and what implications does that have for distributed systems constructed on top of the Tor network. Chapter 5 investigates the impact of creating multiple onion services at the same client to have multiple unlinkable onion service addresses available. The main focus of this chapter will be on the time and resources required to establish new onion services. This was necessary because there was no research data available on this topic and lengthy onion service creation times could delay registration requests from PIAs to a point where the owner of the PIA has already been detected by the sensor. Chapter 6 will investigate the structure of the Tor network as a whole with a focus on the decisions made by the directory authorities. With the decision of Tor's directory authorities being essential to the functionality of the Tor network and thus also to every privacy preserving distributed system using the Tor network, this seemed necessary to ensure that trust in the Tor network is justified. Chapter 7 will utilize the knowledge gained from the previous chapters to introduce several changes to the Tor application that would enable it to better support distributed applications running on top of the Tor network. While all suggested improvements are useful within the proposed networking scheme, most should be beneficial for a range of different systems constructed on top of the Tor network.

Chapter 4

Monitoring the HSDir

This chapter describes an experiment conducted to find out how much information an active adversary with limited resources could obtain about the usage of Tor onion services by controlling a fraction of the hidden service directory. Parts of this research were already published at FOCI'21 [60] and at the official blog of the Tor project [58].

4.1 Preparation

Analyzing the differences between version 2 [126] and version 3 [127] onion services reveals that the Tor project has already gone to great lengths to prevent information leaks from the hidden service directory. Nevertheless, the distributed and anonymous nature of the HSDir makes it impossible to fully prevent information disclosure. In order to quantify the negative privacy impact of the hidden service directory when using Tor onion services, a group of Tor relays was deployed within the Tor network with the explicit goal of collecting information about how onion services are being used. The design of that experiment was inspired by previous research from Owen and Savage [102] that conducted a similar study for version 2 onion services. The basic idea behind this approach is very simple. Deploy several Tor relays that qualify for the HSDir flag and modify them to print a log messages if they receive a request to publish or download a service descriptor. Aggregating the information from multiple relays over extended periods of time should then reveal valuable information about how onion services are being used.

4.1.1 Ethical considerations

Ethics are always an issue if research is conducted on systems that are being used by individuals that did not agree to be part of the research. This becomes even more important if research is conducted on an anonymization network like Tor because there might be users whose life or safety depends on them remaining anonymous. Therefore, research has to be conducted carefully to not endanger any Tor users. Thankfully, the Tor project does operate their own research safety board [121] that supports researchers in conducting experiments on the Tor network safely. One issue discussed in section 3.5.2 is that nodes within the hidden service directory can learn when service descriptors for specific services are being downloaded. Unsurprisingly, the Tor research safety board agreed that collecting this information puts Tor users at risk and recommended that the exact timestamps of upload and download requests should not be stored. They provided many other guidelines as well that significantly improved the experiment. The following sections describe the final experiment setup including all the protections recommended by the research safety board.

4.1.2 Technical Details

For this experiment, we deployed a set of 50 Tor relays (family: `008196DC449482C73CFA9712445223917F760921`) which meet the requirements to obtain the HSDir flag and log every upload and download of a V3 service descriptor by adding two new log statements (see Listing 4.1 and 4.2) to the Tor source code. Instead of writing those logs to disk, they are directly handled by a log listener attached via the control protocol [122] which extracts relevant information from the descriptors, sorts them alphabetically and stores it in a SQL database. Just like the relays, this database was operated by us within our own network on our own hardware. For every upload we store the blinded public key, the relay that received the upload, and a timestamp that only contains the year, month, day and hour of the upload. For downloads we store the same data, except that the timestamp are reduced to daily granularity.

Listing 4.1: Logging upload requests — hs_service.c

```
176 log_info(LD_REND, "New blinded public key: %s",blinded_pubkey);
177 /* Store the descriptor we just got. We are sure here that either
178 * we don't have the entry or we have a newer descriptor and the
179 * old one has been removed from the cache. */
180 store_v3_desc_as_dir(desc);
```

Listing 4.2: Logging download requests — dircache.c

```
1367 /* After the path prefix follows the base64 encoded blinded
1368 * pubkey which we use to get the descriptor from the cache.
1369 * Skip the prefix and get the pubkey. */
1370 tor_assert(!strcmpstart(url, "/tor/hs/3/"));
1371 pubkey_str = url + strlen("/tor/hs/3/");
1372 retval = hs_cache_lookup_as_dir(HS_VERSION_THREE,
1373 pubkey_str, &desc_str);
1374 // Requested public key
1375 log_info(LD_DIR, "TTH: Received Request for Descriptor "
1376 "with ID: %s", pubkey_str);
```

4.1.3 Privacy considerations

In this section we describe the various privacy risks faced by the experiment as well as the solutions applied to mitigate those risks.

Data Management

Access to the machines involved in the experiment was exclusively granted to the members of our research team responsible for the experiment. The database with the collected information will be retained as long as our research is ongoing. Once the research is concluded, the raw data will be erased permanently.

Traffic correlation

On a basic level, our stored data reveals which requests were made to our relays at which time. This might allow an attacker to combine this information with other data sources to launch time-based traffic correlation attacks. To mitigate this issue, we decided to drop all metadata about the requests and to truncate timestamps at the hour value for all uploads. For downloads we saw a greater risk of attack, because they have to be triggered by clients manually, while uploads happen automatically and semi-regularly. Therefore, we decided to cut upload timestamps at the day value instead of the hour. This should render our data useless for time-based correlation attacks, without impacting statistical significance.

Another potential source of correlation raised by the Tor research safety board is the order in which blinded public keys are entered into the database. If that order was the same in which the data was received, this would give attackers an alternative way to accurately determine the time at which certain requests were made. This is mitigated by sorting blinded public keys alphabetically before inserting them into the database.

4.1.4 Hardly used onion services

Some onion services are used for very specific tasks that require only a single user making occasional connections. If one of our relays is a responsible HSDir for such a service and the client selects our relay to request the service descriptor from, our data reveals when an onion service was used. Paired with knowledge about the purpose of an onion service belonging to a single user, this alone might reveal more information about the user than we intended. Unfortunately, we have no way of knowing this in advance, so we cannot exclude such cases during our raw data collection. Consequently, we limit access to our raw data to the researchers responsible for the experiment and make sure that in publications only aggregated information on barely used onion services is published.

4.1.5 Unwanted attention

While our collected information does not contain any onion addresses, attackers with knowledge of onion addresses could easily link our collected blinded keys to addresses they know about. This allows them to use our data to estimate the popularity of an onion service. Since one of the goals of onion services, is keeping the number of users private [51], making this information publicly available could violate the privacy of some onion service operators. Ultimately, this violation could lead some attackers to specifically target onion services because of their popularity within our data.

While this could have been avoided entirely by not storing blinded public keys, we decided to include them in our raw data to enable research on the usage of well known onion addresses and their development over time. This development is especially interesting in the current transition from V2 to V3 onion services. To prevent abuse of our data we will never publish blinded public keys directly (since we have no way of knowing if anyone else will link them to an onion address) and only publish information on how many users an onion service has, if that does not constitute a risk to that service.

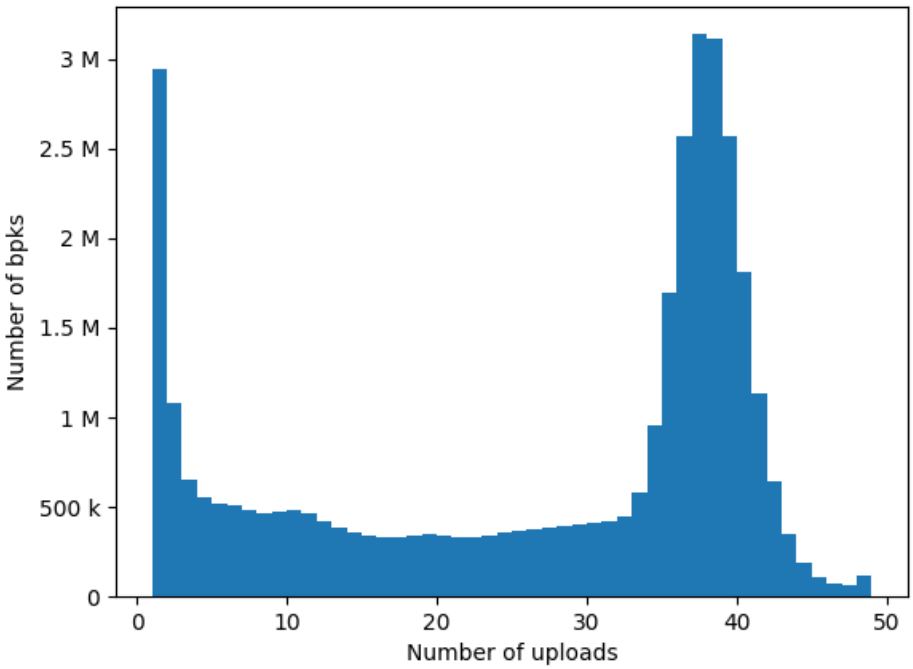

Figure 4.1: How often are bpks uploaded

4.2 Results

While the relays are still running and collecting more data, the collected data is sufficient to answer several questions surrounding V3 onion services.

4.2.1 Uploads

The first interesting piece of information to know about V3 onion services is their average lifetime but, in contrast to previous studies on V2, we have no way of finding out if two blinded public keys belong to the same onion address. However, we do know that similar research on V2 onion services [102] found that most onion services did not live long enough to show up in their data multiple times. While we have no way of confirming these results for V3, we do know that every blinded public key is valid for 48 hours and is re-uploaded at least every 60-120 minutes [127]. Based on this we expect to see every blinded public key uploaded on average 32 times if the Tor network remains stable. Relays joining or leaving the HSDir can cause us to see fewer uploads and unstable introduction points increase the number of observed uploads. Figure 4.1 shows that during our experiment the amount of descriptors seen between 3 and 35 times was fairly constant, which indicates that these were caused by the dynamics of the hidden service directory. The spike at 38 and 39 uploads per BPK seems to indicate that this is the average number of uploads for a stable onion service. The high amount of BPKs with a single upload supports the theory that there are a lot of onion services that live for a very short time. We speculate that these instances are either created by people experimenting with onion services or that there are onion services which are only used once (for example by OnionShare [80]) and thus have no need to republish their descriptor.

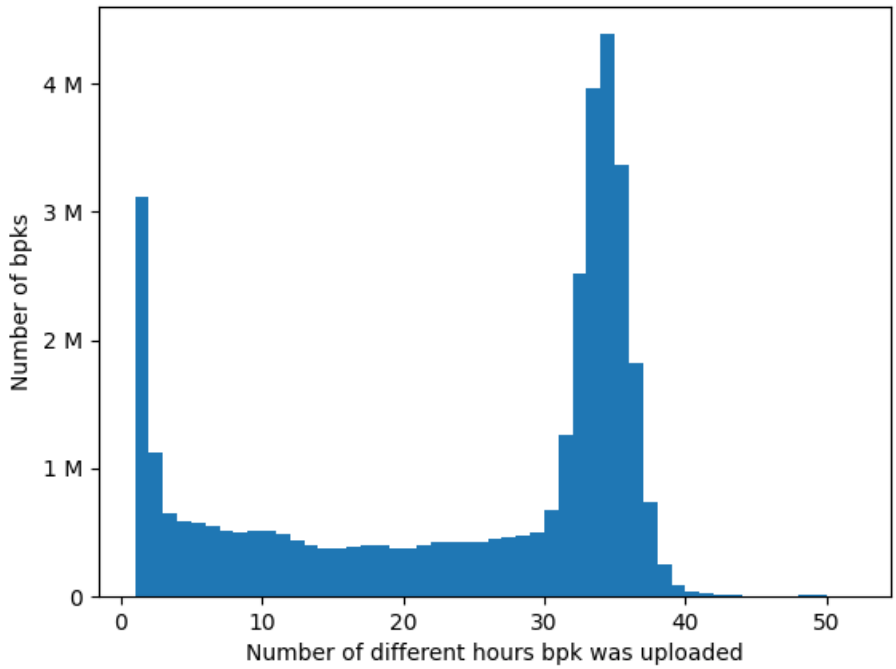

Figure 4.2: For how many different consensuses are bpks uploaded

It should be mentioned that we did detect a small number of blinded public keys that were uploaded more than 50 times to our relays with the record holding key being re-published 16951 times within 48 hours. But since more than 99.5% of all BPKs were uploaded between 1 and 50 times, we feel confident that we can ignore larger numbers as symptoms of misconfiguration of the responsible Tor client.

This caused us to wonder if there are V3 onion services that publish the same BPK over more than 48 hours. Figure 4.2 shows in how many different hours (consensus freshness periods) our observed BPKs were uploaded. In this case we did not have to exclude outliers for plotting since no blinded public key was published more than 53 times.

Comparing Figure 4.1 and Figure 4.2 we can see that we get much closer to our expected 32 uploads on average if we ignore re-uploads in the same consensus freshness period that were most likely caused by changing introduction points or issues with the Tor client of the onion service. The delta between Figure 4.1 and Figure 4.2 can therefore be interpreted as a rough estimate for how often onion services have to re-publish their onion service for other reasons.

The main question we would like to address in this section is how many distinct V3 onion services are available within the Tor network. Before discussing it in more detail, it should be noted that since version 0.4.6.1-alpha[1] Tor also collects metrics about the number of V3 onion service, but no statistics had been published by the time of the FOCI'21 publication [60] and blog post [58]. Since Tor Metrics [125] has in the meantime started publishing their own estimates which highlighted an error in our original calculation, this section will first present the original incorrect estimate and later provide an extended discussion of the corrected estimate.

[1]https://blog.torproject.org/node/2011

To make this comparison as easy as possible, we adapted the specification on how Tor Metrics generates statistics on V2 onion services [51] and only made the necessary changes to be compatible with V3. First, we need to know the fractions of descriptors seen by our members of the hidden service directory based on the hash value ($hsdir\_hash_x$) that determines their position within the HSDir. Like regular Tor clients we can derive the HSDir from a Tor consensus [127] and calculate the fractions of our own relays as

$$hsdir\_share_x = (hsdir\_hash_x - hsdir\_hash_{x-4}) \,/\, 2^{256} \,/\, 4.$$

Dividing *hsdir_share* by 4 is necessary to counter the effect that every service descriptor is uploaded to four consecutive relays. To estimate the total size of the hidden service directory, we need to combine our relative share with the number of uploaded blinded public keys (bpk_count_x):

$$extrapolated\_size_x = (bpk\_count_x \cdot (1 \,/\, hsdir\_share_x)) \,/\, 4.$$

The division of the *extrapolated_size* by 4 is necessary to obtain the number of distinct V3 onion addresses since every service maintains two replicas in two time periods. Unfortunately, this is where our calculation was wrong. We believed that dividing the share by four would accommodate for the service descriptor being uploaded to four different nodes within the HSDir, but this is not the case. Dividing the *hsdir_share* by 4 does not reduce the estimated number of unique onion services, it increases it by a factor of four because every onion service only sees a quarter of the uploads within its share. Extrapolating the number of *unique_bpk_uploads* and the *estimated_size* should therefore be done like this:

$$\begin{aligned} unique\_bpk\_uploads_x &= bpk\_count_x \cdot (1 \,/\, hsdir\_share_x) \\ extrapolated\_size_x &= unique\_bpk\_uploads_x \,/\, 16 \\ &= (bpk\_count_x \cdot (1 \,/\, hsdir\_share_x)) \,/\, 16 \end{aligned}$$

To aggregate this set of extrapolated sizes to a single value, we opted to use the weighted interquartile mean [51] that the Tor metrics team uses to estimate the V2 onion address count. Figure 4.3 shows the results of our original calculations which indicated that the number of V3 onion services was between 600,000 and 700,000 in March and April of 2021 making them about three times more popular than V2 onion services. Thankfully, the missing division by four can be easily applied retroactively to conclude that the real estimate should have been between 150,000 and 200,000 instead. The drop to zero between March 18th and March 23rd was caused by a temporary outage of the directory authority *Faravahar*, which caused all of our relays to lose their HSDir flag until the authority came up again. This event triggered a closer investigation into the Tor consensus, which is presented in chapter 6. An important criterion to judge the significance of our measurements is the relative share of the hidden service directory we are extrapolating from. Tor metrics define a threshold of 1% for V2 onion services and excludes data for days with less information. Figure 4.4 shows the relative share of the hidden service directory we monitored during our experiment. Although our share was quite volatile jumping between 0.8 % and 1.2 %, our average observed HSDir share was meeting the margin set out by Tor metrics to obtain relevant information and should therefore provide useful results.

When Tor Metrics began to publish their own estimate on the number of V3 onion services, their estimate was obviously much lower than our original estimate, leading to the discovery of our missing division by 4. Yet, even after correcting for this error, our estimates did still not correspond to the numbers

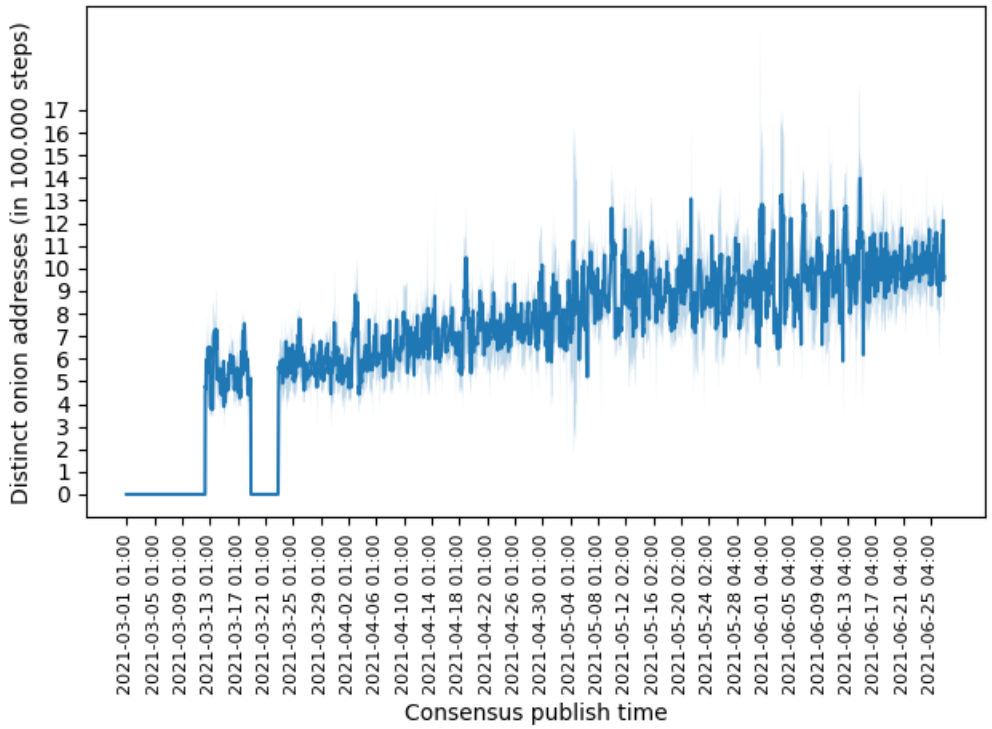

Figure 4.3: Incorrect extrapolated number of V3 onion services (Published at FOCI'21 [60] and the Tor Blog [58])

provided by Tor metrics, leading us to conclude that there were additional errors within our calculation. To identify our remaining issues, we downloaded the publicly reported statistics on V3 onion service usage and tried to reproduce the estimates published by the Tor Metrics team. Our attempts were initially unsuccessful, until we realized that we were not the only ones with errors in our calculation. The Tor Metrics team has not considered the fact that the V3 onion service specification introduced a new way to create the HSDir.

$$hsdir\_share\_v3_x = (hsdir\_hash_x - hsdir\_hash_{x-4}) \,/\, 2^{256} \,/\, 4$$
$$hsdir\_share\_v2_x = \left(fingerprint_x - fingerprint_{x-4}\right) \,/\, 2^{160} \,/\, 3$$

V2 onion service used to calculate their share ($hsdir\_share\_v2_x$) based on their static fingerprint, allowing attackers to predict the future position of a relay within the hidden service directory. Since this was successfully exploited by researchers [11], Tor made the position of V3 HSDir relays less predictable. This change was not taken into consideration by the Tor Metrics team, causing them to extrapolate their estimates like this:

$$extrapolated\_size\_metrics_x = (bpk\_count_x \cdot (1 \,/\, hsdir\_share\_v2_x)) \,/\, 16.$$

Obviously, this is also incorrect but assuming that no attacker is intentionally deploying relays with specific positions within the hidden service directory, the shares might be similar enough to have negligible impact on the final estimate. What definitely has an impact however, is the fact that the estimate is only divided by 3, which was correct for V2 onion services as they were only uploaded to 3 HSDir nodes per replica instead of 4. In order to confirm that this was indeed the explanation for the differences between the estimates, we calculated

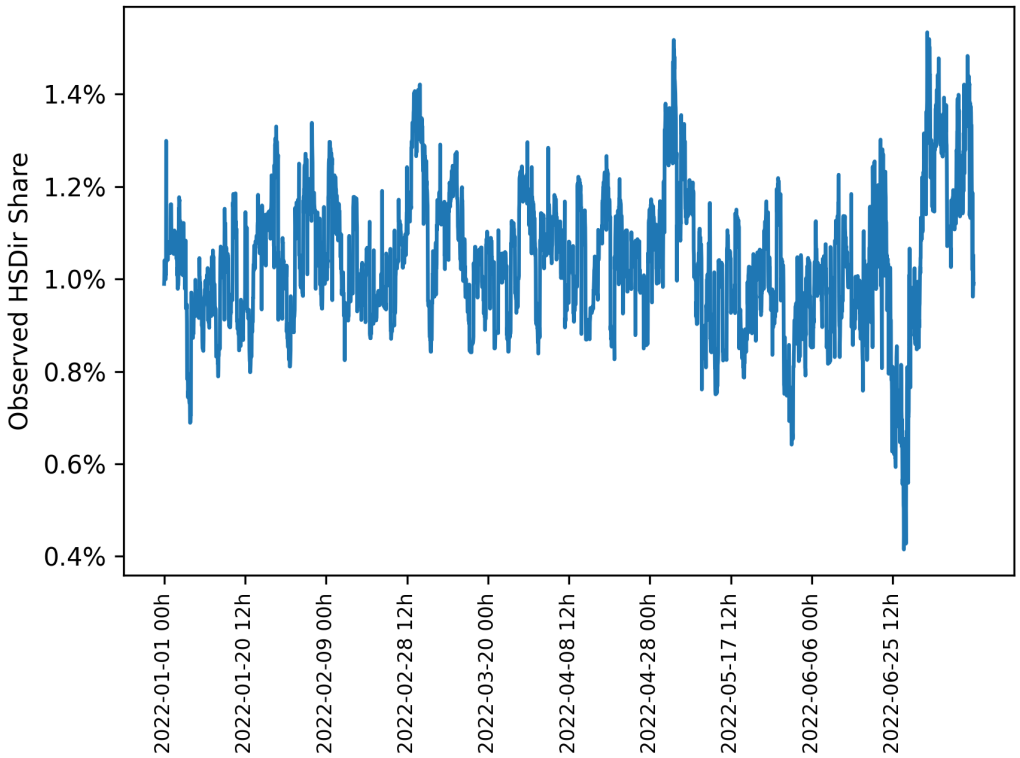

Figure 4.4: Relative share of HSDir we observed

the HSDir shares for V2 and V3 onion services and used both of them for extrapolation. Figure 4.5 confirms our assumption by proving that our algorithm reproduces the results by Tor metrics almost perfectly while using *hsdir_share_v2* to extrapolate network totals. The same graph also shows where the estimate would end up, if *hsdir_share_v3* were to be used instead. Unsurprisingly, the main factor seems to be the division by 3 instead of 4 resulting in estimates calculated from V2 shares being on average 20 % lower than they would be for for V3. The second graph now uses the data gathered by our experiment instead of the officially reported data. To make comparisons with the data collected by the Tor project easy, we aggregated our data by the same 24 hour periods that are used by the Tor project. While the variance of our estimates is much larger than it is for the official data, the overall estimate remains similar enough to convince us that our data is representative of the HSDir. At this point we also decided to inform that Tor Metrics team about our findings and they confirmed our conclusion that the current estimate for unique V3 onion addresses is incorrect because it uses the HSDir shares for V2 onion services [57].

With both our implementation and our data verified against the results from a second source, it is worth to discuss a few potential issues with the current way of estimating V3 onion service usage. The first thing to address is that official data reports only the number of unique *blinded public keys* observed by a single member of the HSDir during a 24 hour period. During analysis, it is assumed that those BPKs were evenly spaced across the 24 hour aggregation period. Since our experiment is keeping track of uploads per hour, not per day, we can actually verify this assumption. Figure 4.6 shows the estimated number of onion services for every single hour/consensus within the observation period. Note that this hourly estimate must be analyzed with great care. We know from

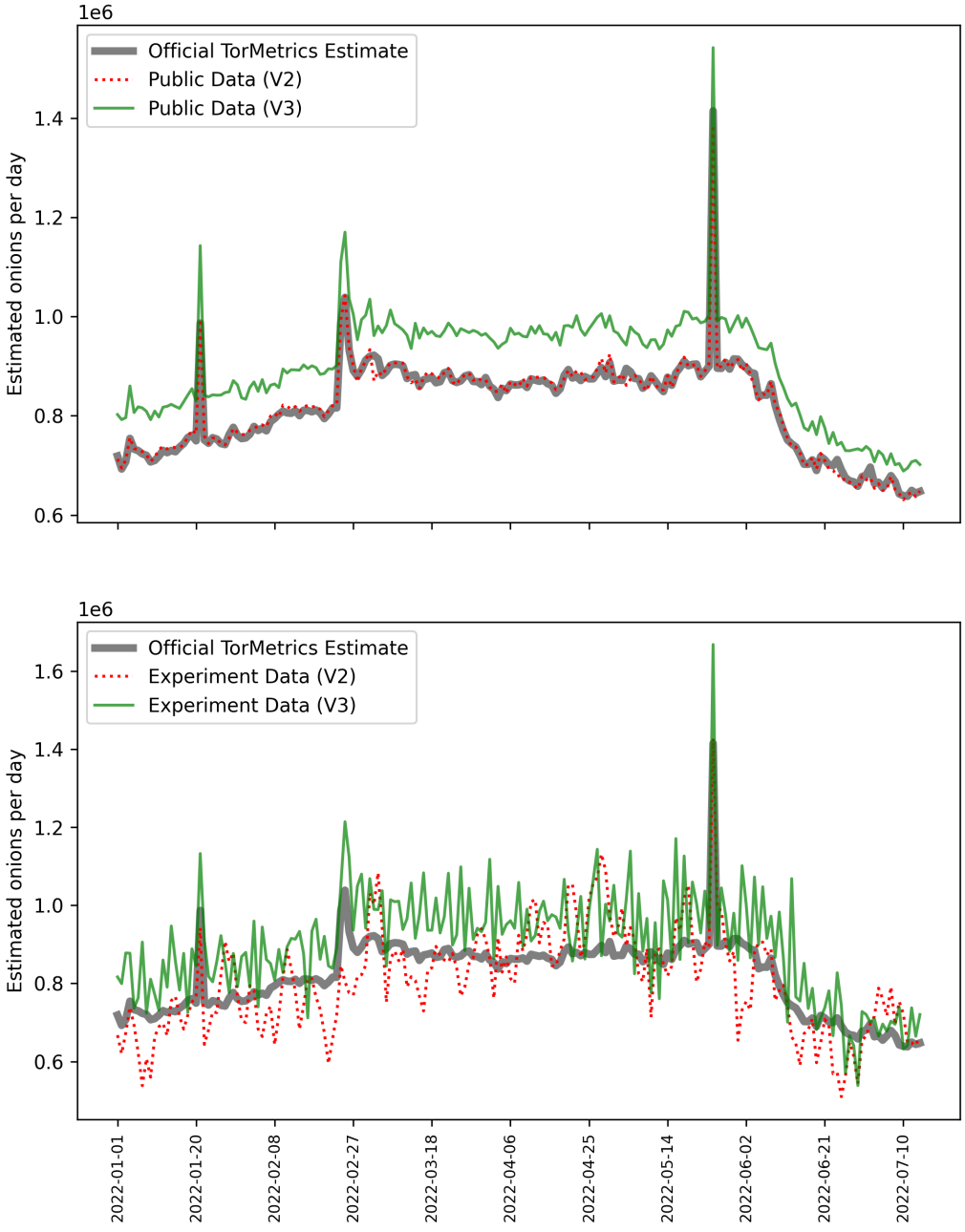

Figure 4.5: Comparing estimates on number of distinct onion addresses

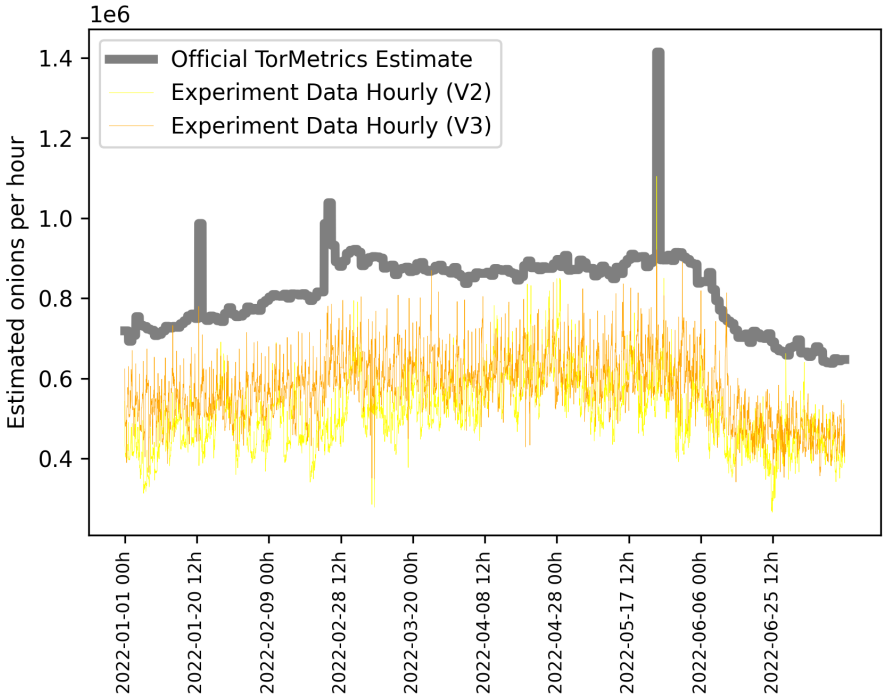

Figure 4.6: Estimated number of distinct onion addresses per hour

Figure 4.2 that BPKs are typically only uploaded to about 32 of 48 consensuses, which causes the estimate to be lower than it should be. At the same time, there is a substantial share of BPKs that is only uploaded within a single hour which are not taken into account during aggregation. The estimate of running onion services provided by the Tor network is therefore always higher than the actual number, because not all onion services included in the 24 hour aggregation period have been active at the same time. The more interesting observation to make when investigating the daily estimates however, is their correlation with the spikes in the officially reported data. The most significant spike happens on the 05-25-2022, where the estimated number of V3 onion services jumps from less than 1 million to more than 1.4 million for a single day. Our collected data contains the identical spike when aggregated over 24 hour periods, but in the hourly aggregation we realize that this spike was caused within just three hours (between 00:00 and 03:00 to be precise).

4.2.2 Downloads

In order to assess the relevance of onion services, it is essential to estimate the amount of users they are handling. There are no official statistics on how many users onion services have collected by the Tor network and previous research [11, 84] has focused on what onion services are used for, so there is little data [102] on the usage of onion services and it only applies to version 2 onion services.

We can provide some insight into how frequently onion services are being accessed by investigating the number of times every blinded public key was requested from our nodes. It should be noted that descriptor downloads do not

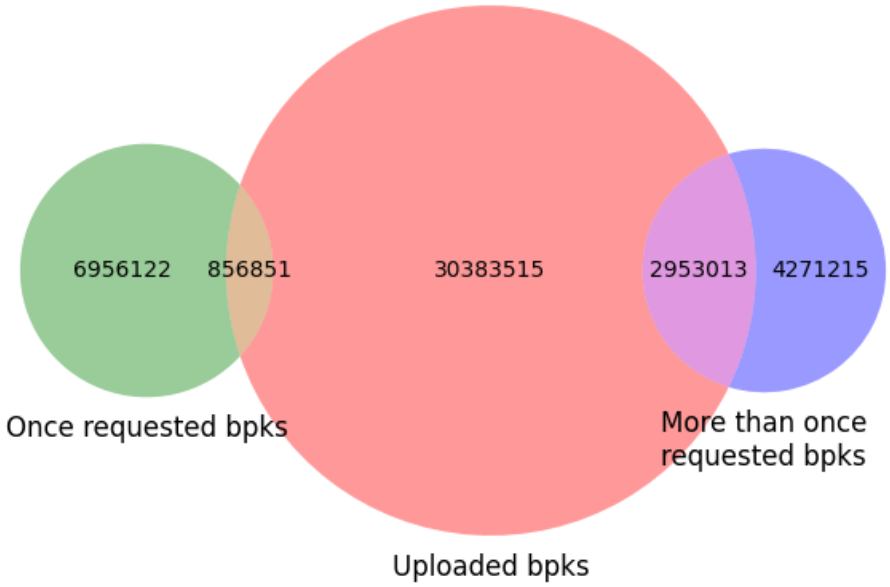

Figure 4.7: Relation between uploads and downloads

correspond to visits since the Tor client caches descriptors, but also not to visitors since there are several reasons that could cause Tor to request a descriptor multiple times. Even more specific, the download of a descriptor only tells us that a Tor client intended to connect to an onion service, not if it actually connected. So we interpret the count of descriptor downloads as an upper bound on the number of visitors and a lower bound on the number of visitation attempts.

Tor clients use only 6 out of the 8 responsible HSDir relays for downloading (the other two are there in case a relay goes offline). Usually, one of those 6 relays is chosen at random, so our recorded number of descriptor downloads must be multiplied by a factor of six to extrapolate the actual number of descriptor requests for a specific blinded public key. The only exception to this rule are requests for onion services that no longer exist. In this case a Tor client will try all six responsible hidden service directories before giving up, so there is no need to extrapolate the number of download attempts.

Since we already learned from the uploads that many onion services have a very short lifetime we expected a high number of onion services with a very low download count. Figure 4.7 confirms this theory by showing that the vast majority of uploaded blinded public keys is never downloaded. When interpreting this result, it is important to remember that we usually only control one out of eight responsible hidden service directory nodes, so an unused upload does not mean that the service was never accessed, it just means it was never downloaded from our node. V3 descriptors are always uploaded to 8 different nodes, but only downloaded from 6 (the other two serve as backup if a HSDir node goes offline), so about 25 % of our uploads are meant to be unused. The remaining unused uploads are most likely caused by onion services that are either unused or used so rarely that our node was never chosen by chance and remained un-

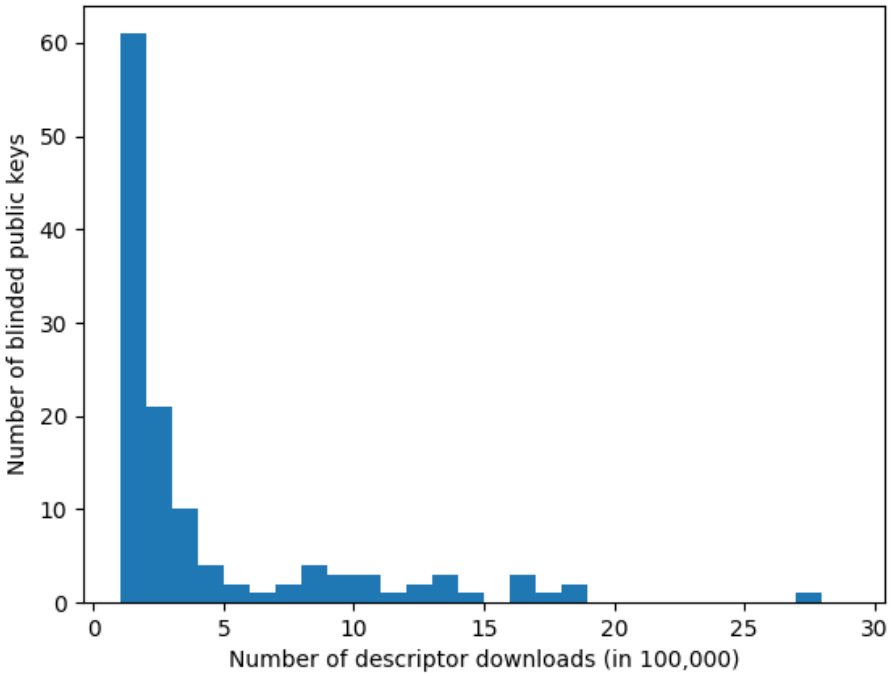

Figure 4.8: Top 0.01% of most downloaded bpks

used. To get an idea for how often this might happen, Figure 4.7 distinguished between keys that were only downloaded once and keys that were downloaded more than once. This should provide a rough estimate for the difference between onion services with a single user and onion services with multiple users. However, even when accounting for 25 % backup descriptors and not seeing 83.3 % of once requested BPKs, we are still left with more than half of our received uploaded blinded public keys never being downloaded. This leads us to conclude that the vast majority of onion services is either not used at all or at least not on a daily basis. A fairly surprising result of our analysis was the small overlap between uploaded and downloaded blinded public keys. Less than 11 % of the requested blinded public keys that were only requested once had been uploaded to our servers. For blinded public keys that were requested multiple times, this share increases to slightly above 40 %, indicating that the vast majority of download attempts made be Tor clients actually results in failure. A potential explanation for infrequent download attempts on onion services might be crawlers that continue trying to access onion services even after they have been disabled. Multiple crawlers running independent from each other could also explain some of the failed download attempts for more popular onion services.

Next, we took a look at the most downloaded blinded public keys to quantify how successful the most popular onion services are. We found that more than 77.5 % of all download requests received by our relays were asking for the most popular 1 % of blinded public keys with the record key being requested more than 1.6 million times within 48 hours. This implies that around 9.6 million attempts to visit the service were made during one day. Figure 4.8 shows the top 0.01 % of most downloaded blinded public keys which make up 47 % of all downloads for blinded public keys. This illustrates nicely that a very small

number of onion services is responsible for the majority of onion service usage. While we have no way of knowing what kind of onion service would be so popular, one idea was that some botnets might be using onion services as their command and control server. Previous researchers have already found that V2 onion services were used to control botnets [102] and it would not be a surprise if V3 onion services continue to be used in this way. A possible way to support this theory is to quantify the number of requests for blinded public keys that were never published. In contrast to humans, programs tend to repeatedly try to connect to a no longer working onion service while humans will give up after a short time. In our data set, there are several blinded public keys that were never published to our relays, but still requested more than one million times—and we only observe 12.5 % of the total download attempts—over 48 hours. We see two possible explanations for this observation: Either these keys belong to a defunct botnet server or they were victim of a DoS attack that knocked out the onion service in the previous time period but did not stop trying to attack.

Figure 4.7 also shows that our relays could only respond to 25.3 % (3809864/30383515) of all descriptor download requests with a valid descriptor. The remaining 74,7 % of requests were asking for blinded public keys that were never uploaded to any of our relays. Since we do not record the exact timing of requests, the number of failed requests is likely to be even higher because blinded public keys might already be requested before they are being published. While there is a low possibility that regular running onion services might cause this behavior, this requires very specific circumstances: The onion service would have to publish their descriptor shortly before a new consensus is published, that new consensus would have to change the HSDir so that a new HSDir relay becomes responsible for the descriptor, then the relay must random a long enough descriptor interval period to never upload during this consensus interval, and finally the next consensus would have to revert the change back to a state where the HSDir relay that became responsible for the onion service is no longer responsible for the onion service. In this scenario, clients request the blinded public key of an onion service from a HSDir relay that is completely unaware of the onion service although the onion service is up and running. As this scenario cannot reasonable explain the large share of failed requests, we conclude that a significant share of requested onion addresses belongs to disabled/nonexistent onion services. The distribution between available and unavailable onion services also applies to the blinded public keys depicted in Figure 4.8 meaning that a majority of onion service requests is asking for a small number of onion services that often do not exist.

The fact that so many onion service requests fail paired with the justified assumption that many high-volume onion services are apparently used for machine-to-machine communication means that we are unable to derive any conclusions about how many human users connect to onion services.

4.3 Summary

This experiment provided several valuable pieces of information about the Tor network. It confirmed our assumption that the Tor network is capable of sustaining a vast amount of onion services ranging from services that receive millions of requests every day down to service that regularly see no clients at all. The observation of onion services being requested frequently without ever being published indicates that there are already a lot of systems that build on top of the Tor network and use Tor onion services for communication. This encourages continued work on Tor onion services to see if they can be optimized

for Tor-aware privacy preserving applications that route their entire network communication through the Tor network. The significant amount of information that could be collected by a small adversary with access to only 25 IP addresses also confirms the original assumption that the hidden service directory constitutes a threat to the privacy of Digidow's users, if their PIAs use default Tor onion services. Further improvements will be necessary to make the networking approach presented in section 3 resilient against such attackers. At the same time, the Tor project should most likely continue to investigate onion services that cause millions of descriptor downloads, especially if the downloads are looking to retrieve descriptors that do not exist. Understanding what they are and removing their useless requests would most likely improve the Tor network as a whole.

Chapter 5

Short-lived onion services

As discussed in section 3.6.5, PIAs need to use different onion addresses to prevent multiple sensors from realizing that the same individual has registered at all of them. Although Tor does allow Tor clients to create an arbitrary number of Tor onion services, there was no published research available that investigated the time needed to create onion services dynamically and the number of onion services that a single Tor client can reasonable operate simultaneously. The research presented in this chapter closed this gap when parts of it were published at WTMC 2021 [59].

5.1 Experiment Design

To find out if it is feasible to dynamically generate onion services in any scenario, the associated negative performance impact must be properly quantified. To do this, we measure the time between instructing a host to generate an onion service and clients being able to access it. Only if this latency is sufficiently low, it may be acceptable to generate onion services on-demand, with the exact time limits depending on the situation. PIAs for example could precompute onion addresses in advance to prevent the onion service creation time from negatively impacting user experience. Our measurement setup is inspired by previous work of Loesing et al. [87] and Lenhard et al. [82], but instead of measuring the time it takes to access an onion service, we measure the time it takes to create one.

5.1.1 Measurement Setup

We use the Tor Stem[1] library to interact with Tor's control protocol [122] and generate onion services. Timing information is extracted from the log file created by Tor and event listeners attached via the control protocol. This allows measuring the time of the following events:

- Start connecting to introduction point,
- circuit to introduction point established,
- introduction point ready,
- service descriptor created,
- start upload to HSDir, and
- finish upload to HSDir.

[1] https://stem.torproject.org/

No good solution was found to measure the time it takes Tor to select introduction nodes when creating a new onion service. It seems reasonable to assume that this time is insignificant for the overall latency, but it could be speculated that one host running many onion services could experience deteriorating performance as Tor does not reuse introduction points[2].

All our tests were conducted with version 0.4.3.5 of Tor and ran on a virtual machine running Debian 10, which was monitored to ensure that no local limitations regarding CPU, bandwidth, latency, or memory would impact our measurements. The Internet connection (1GBit/s, low latency) was constantly monitored to be working within "normal" parameters, in order to assure that we do not accidentally measure latency effects or outages introduced primarily through our own Internet link.

To ensure that measurements do not influence each other, a new Tor process is completely bootstrapped within a fresh Docker container for every onion service. Our test system runs one test at a time to avoid different onion services impacting each other. To mitigate the effects of possible issues with our Internet connection or the Tor network, tests are conducted in a loop. Every iteration of the loop tests every configuration once. This loop ran more than 1500 times over a period of 10 days to obtain a sufficiently large sample size.

The container specification and our measurement implementation are available at https://github.com/mobilesec/onion-service-time-measurement to enable other researchers to reproduce our measurements.

5.1.2 Measured Configurations

As already mentioned, onion services are still in development and could, therefore, at the time of the experiment be deployed in different configurations. To find out if the method of deployment has an impact on the provisioning time of onion services, four different types were measured:

1. *V2*: A V2 onion service with default parameters: Old, no longer recommended version, which was mainly included to enable comparisons with previous research. V2 onion services were not yet deprecated when this research was conducted.
2. *V3*: A persistent V3 onion service with default parameters.
3. *Ephemeral*: A V3 onion service with default parameters which can only be created via the control protocol and will only exist as long as the control connection to the Tor instance is maintained.
4. *Vanguard*: A V3 onion service with the Vanguard [6] extension to harden it against different deanonymization attacks.

5.2 Results

Figure 5.1 provides a good summary of the results of our analysis. The changes implemented by V3 of the onion service protocol have significantly improved deployment times from about half a minute to less than 10 seconds. There are no significant differences between normal and ephemeral onion services, which is no surprise considering that the only difference between those is

[2]The official documentation still has an open TODO on picking nodes. However, a review of the Tor implementation revealed that this is the case.

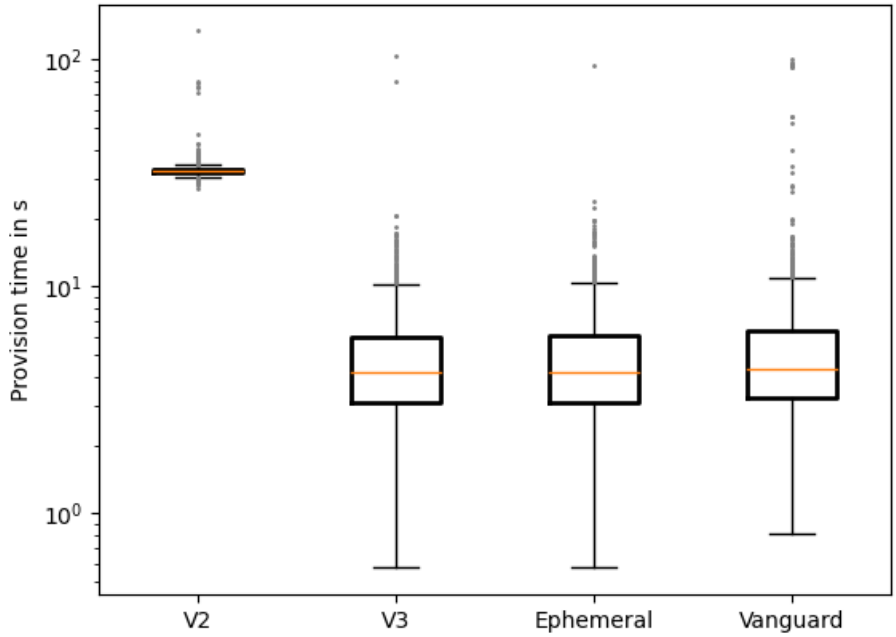

Figure 5.1: Overview of provisioning times

the persistence of cryptographic keys on disk. The Vanguard extension also shows no significant changes in provisioning time, which is unexpected because modifying Tor's behavior via the control protocol should actually cause a performance overhead, but is apparently not relevant for our measured scenario.

5.2.1 Provisioning Stages

A potentially interesting explanation for the significant differences in provisioning times between V2 and V3 is provided by figure 5.2. It splits the provisioning into three stages:

1. The time it takes the host to establish the introduction points for the onion service,
2. the time it takes to generate a descriptor for uploading after introduction nodes have been established, and
3. the time it takes to actually upload the descriptor.

The first fact to note here is that V2 onion services appear to take much longer to generate their service descriptors. Since there was no obvious reason for such a significant performance difference, we investigated the source code and found that the current implementation of Tor V2 onion services waits 30 seconds before uploading a descriptor. There is no explanation in the specification [126] as to why this delay is necessary and the source code only comments that the delay is introduced to ensure that the descriptor is stable. Since V2 onion services were disabled in October 2021 [47], we did not spend additional time on analyzing this issue.

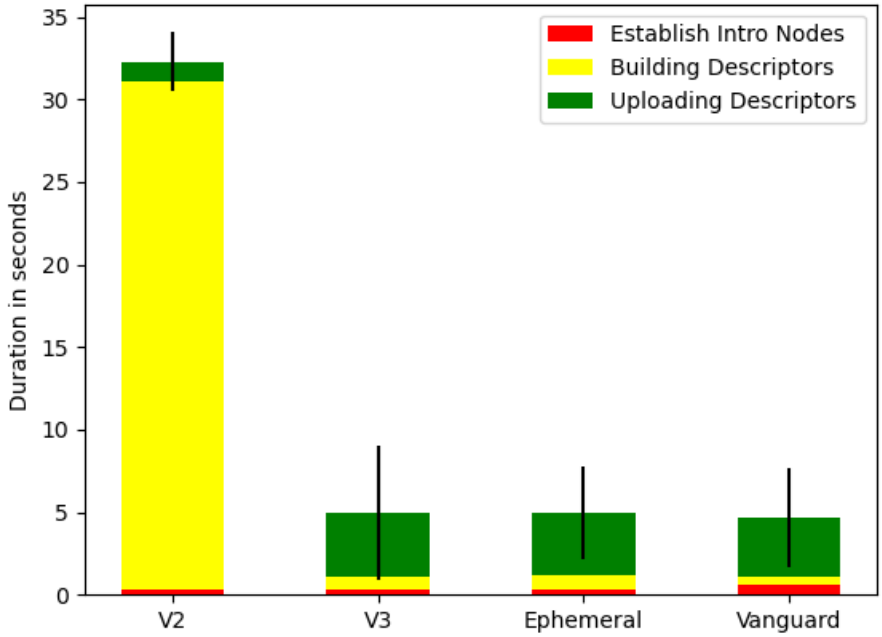

Figure 5.2: Composition of provisioning times

Figure 5.2 also reveals other less obvious, but interesting, aspects. For example, it confirms a suspicion that is hard to verify on the logarithmic scale of figure 5.1, namely the fact that for V2 onion services, upload times have not only less impact on the total provisioning time, but are also lower in absolute numbers. The exact reasons for this behavior is analyzed in section 5.2.2.

Another interesting observation is the fact that establishing introduction nodes is insignificant to the provisioning time of an onion service across all configurations. This observation is however not fully correct because, as already mentioned, all our measurements were conducted with fully bootstrapped Tor instances. During the bootstrapping process, several circuits (in our experiments we encountered between 2 and 15 circuits during bootstrapping) are prepared, so they can be used for later connections. In our setup, these circuits are always used to connect to introduction points, so our measured time for the creation of introduction points does not include the circuit creation time. Since Tor already collects detailed metrics on circuit creation time [86, 125], there was no reason to analyze them at this point.

What is worth noting, is that the Vanguard plugin almost doubles the introduction node building time, without impacting the overall provisioning time. At first glance, this seems to imply that Vanguard is actually decreasing the descriptor creation time, which is unlikely considering the fact that Vanguard makes no changes to service descriptors. Instead, the difference is caused by the fact that the generation and derivation of all keys required for creating a service descriptor take a constant amount of time and can already start before the introduction points have been selected. We verified this by deploying onion services with 10 introduction points. Naturally, they needed more time to establish their introduction points, but they still finished creating their de-

scriptors at the same time as services with only three introduction points. This shows that the time needed to establish introduction points is currently irrelevant for the provisioning time of an onion service.

Our final observation is that the descriptor upload is the most significant factor for total provisioning time in current onion service configurations, so we look at them in more detail.

5.2.2 Descriptor Upload Times

Our results for descriptor upload times have to be put in context to be interpreted correctly: Every onion service uploads its descriptor to several nodes on the hidden service directory. The number can be configured by each service, but the defaults are three nodes for V2 descriptors and four nodes for V3. Both are uploaded in two replicas, so in total there are 6 and 8 uploads. Additionally, the V3 onion service specification requires them to always be valid in two time periods, the previous one and the current one. So when creating a new V3 service from scratch (as done by our test setup) 16 descriptors are uploaded initially.

When Tor clients try to access an onion service, they use their current time period. The previous one is only uploaded to avoid synchronization issues with clients that are still in the previous time period. Tor clients randomly pick one out of only three nodes from one of the two replicas. The fourth upload in V3 is only there to handle situations where a HSDir node goes offline. This means that a single upload could be sufficient to allow an incoming connection. Unless there are any issues with synchronization or failing nodes, six uploads already enable full connectivity. Our measurement setup was not designed to take this into account. Instead, we assume that a descriptor has been successfully published when half of all uploads (3 for V2 and 8 for V3) have been completed. The upload time in figure 5.2 shows how long it took on average to complete half of all uploads. This decision removes the impact of very slow uploads and tries to find a middle ground between trying to find the earliest time when connections are possible and the time when connections are almost certain to succeed without retries.

Figure 5.3 shows the duration of individual descriptor uploads. The majority of upload requests finish in less than 5 seconds and almost all uploads complete after 20 seconds. A noteworthy result of our measurement is an unexpectedly high number of upload requests that take between 100 and 105 seconds, which occurs for all measured configurations, but happens less often for onion services with the Vanguards extension. To further analyze this behavior we conducted a second smaller experiment by running the loop only 500 times and additionally tracking the time when upload circuits were completed. This allows us to split the upload time into the time it took to create a circuit and the time it took to actually upload the descriptor.

Figure 5.4 shows that plain uploads hardly ever exceed five seconds and even the slowest single upload we measured only took 12 seconds to complete. The unexplained 100 second delay is only present in the circuit creation time shown in figure 5.5. This makes sense because this delay only happens when Tor fails to open a circuit to a hidden service directory. In this case a 100 second timeout occurs before another attempt is made. This also explains why Vanguards has a positive effect on this issue. It selects a subset of candidate nodes for the second and third hop of a circuit and tries to reduce the risk of picking a malicious node. Apparently, this also reduces the risk of picking nodes that cause circuit creation attempts to fail, increasing the overall performance and reliability.

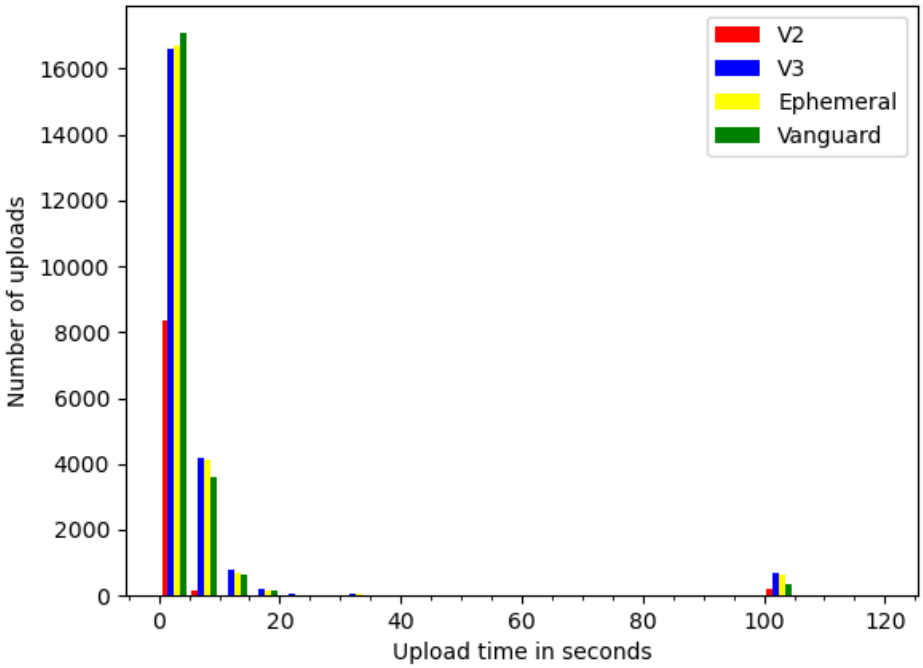

Figure 5.3: Time it took individual uploads to complete

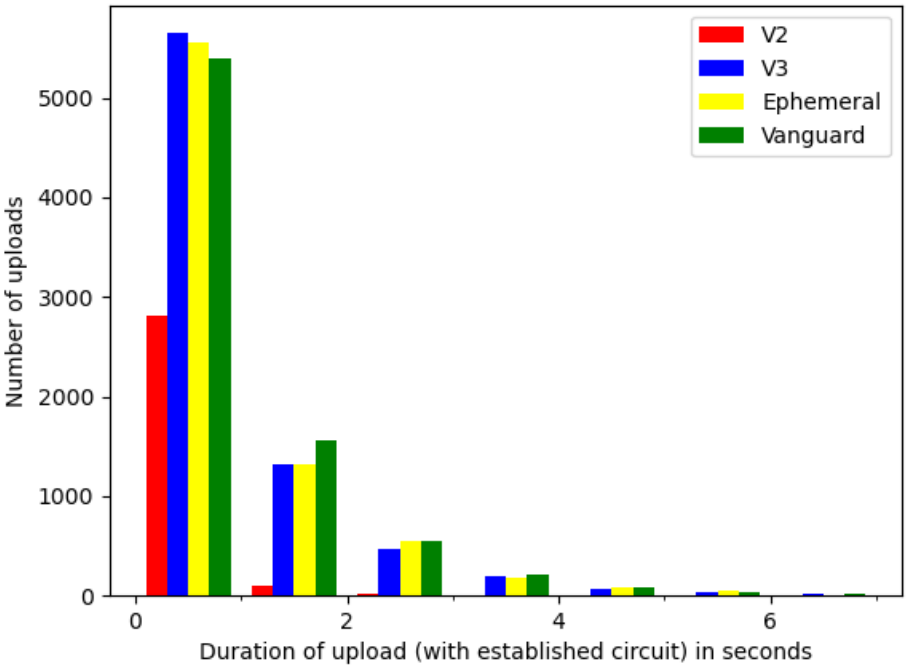

Figure 5.4: Time to upload descriptor via established circuit

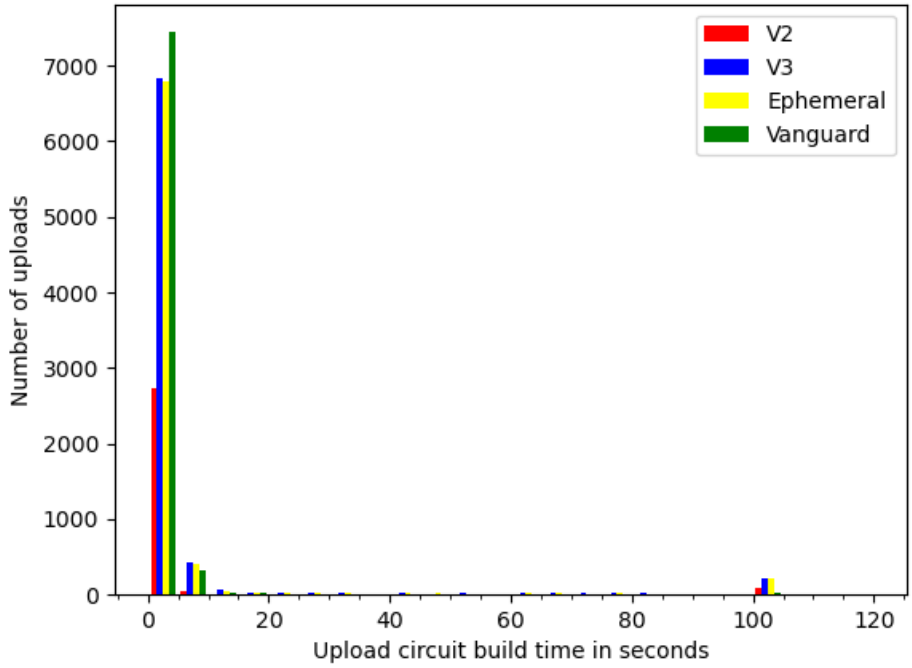

Figure 5.5: Time to create upload circuit

Another interesting result in this context is the fact that some circuits fail again after this 100 second timeout. In this case Tor does not wait and try for a third time, but instead abandons the upload attempt entirely. This does not result in any error displayed to the user, because the onion service concept is redundant and a single failed upload has no significant impact on the availability of an onion service. During our experiments we experienced an upload failure rate of about 1% for upload requests without Vanguard and a failure rate of about 0.8% for uploads with Vanguard.

Figure 5.4 confirms that V2 descriptors are published faster than V3, which is most likely caused by the much larger descriptor size in V3. Figure 5.6 provides a zoom-in on circuit build times below 8 seconds to facilitate comparison with Figure 5.4 and shows that the circuit creation time has more impact on how long it takes to publish a descriptor than the actual upload. An interesting observation is that our results seem to indicate that V2 upload circuits are created faster than V3 upload circuits. This effect is again caused by the fact that our Tor binaries were fully bootstrapped before any measurements were conducted, which allows Tor to cannibalize general circuits for uploads if there are any available. Since cannibalization is much faster than creating a circuit from scratch, this means that some upload circuits can be created faster than the rest. The lower number of uploads in the V2 onion service specification [126] increased the relative impact of these cannibalized circuits, creating the impression that V2 upload circuits are created faster. Unfortunately, we could not properly quantify the impact of this issue, so we cannot say if there are any other factors contributing to the increased circuit creation time in V3.

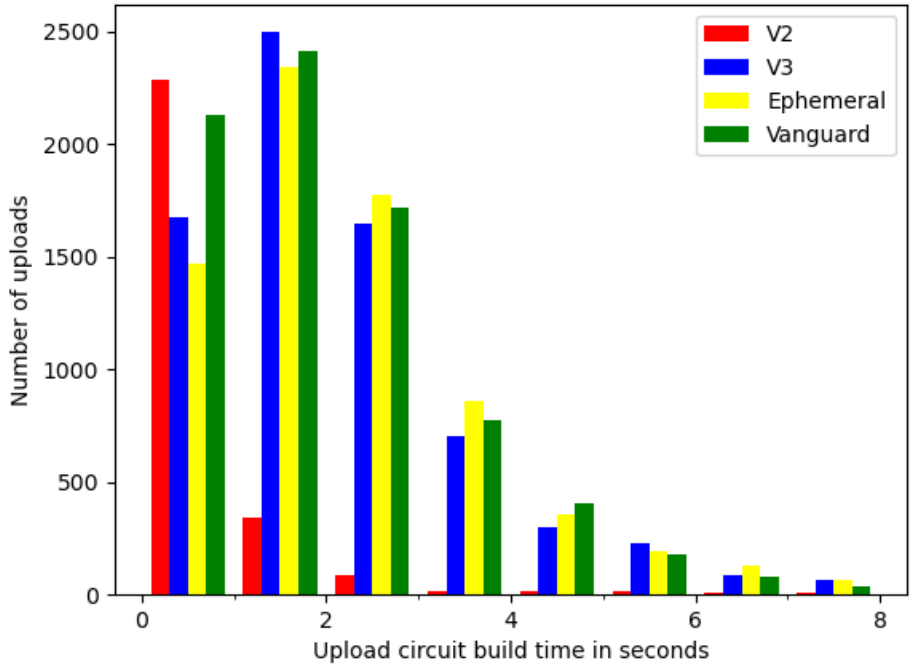

Figure 5.6: Zoom-in on upload circuit build times below 8 seconds

5.3 Summary

While improved performance was not an explicit objective of the V3 onion service protocol, our measurements show that the deployment performance was improved significantly from more than 30 to about 5 seconds. Despite these improvements, onion service deployment still takes several seconds, which is definitely too long if users experience that delay with every onion service creation. Since the callback addresses needed for the *register* messages from section 3.1.5 can be prepared in advance, the creation time might be acceptable, but it raises several new questions: How many onion services should be prepared in advance? Too few and the PIA needs several additional seconds before it can register to all sensors, too many and PIAs might end up putting too much load on the Tor network or draining too much power on mobile devices. A potential solution to put an upper limit on the amount of needed onion services is to adopt the concept of rolling identifiers from the Exposure Notification framework [45] designed to trace contacts during the Covid-19 pandemic. They broadcast a rolling identifier that changes every 15 minutes based on the assumption that adversaries being able to track at most 15 minutes of consecutive movement would not cause relevant privacy risk. While there is no research yet to actually confirm this assumption, it does seem reasonable. PIAs could do the same by using rotating onion addresses instead of dynamic ones. Assuming onion services being rotated every 15 minutes and timeouts of at most 60 minutes, PIAs would need at most 5 onion services at any given point in time and the need for new onion services would be perfectly predictable, removing the need for quick creation of onion services.

An alternative approach that also seems worth further research is based on the

discovery that roughly 80% of the time needed for publishing an onion service is spent on uploading descriptors to the HSDir. If the need to publish an onion service could be removed entirely, for example by including service descriptors in *register* messages instead of onion addresses, dynamic onion service generation becomes a lot more feasible. It shortens the time needed to prepare an onion service to less than one second and reduces the load that the creation of dynamic onion services puts on the Tor network. This might also be interesting for other scenarios that cannot rely on rotating service descriptors, like the numerous projects exploring file sharing [80] or instant messaging [13, 83, 129]. In an instant messaging scenario, an initial exchange of service descriptors could take place when two individuals are in close proximity by encoding a service descriptor in a QRcode and displaying it for the communication partner to scan. Considering that v3 onion service addresses need to encode a key already, they are not particularly easy to type or remember, making it likely that messengers will provide a more convenient way of exchanging contact information already. After the initial contact has been established, clients could append the next service descriptor to sent messages, thus iterating the used onion service with every exchanged message. This would increase the time it takes to send a message, because a new service descriptor has to be created every time, but an additional delay of a single second might be acceptable for privacy sensitive scenarios. User experience could again be further improved by already setting up a new onion service while the user is still typing. Avoiding the hidden service directory entirely has the additional benefit of removing the potential privacy leaks identified in chapter 4.

Chapter 6

Verifying the Tor consensus

6.1 Analysis

During our analysis of the Tor consensus we confirmed that many things work exactly as expected. In this work we will not discuss any confirmatory results and instead focus on unexpected behavior and intriguing observations that we encountered during our analysis.

6.1.1 Data Sources

To analyze and evaluate the decisions made by directory authorities in the past, one needs access to as much information as possible about the Tor network. Thankfully, the Tor team archives [86, 125] all documents made available by clients since 2005, so we do not have to worry about data collection for our research. Instead, we can just use the official data archive, meaning that all of our results and graphs reflect the state of the Tor network according to their own archives.

We have access to all published consensus documents, the votes that were used to create the consensus, as well as the server descriptors that provide extended information about every relay. This data enables us to find out which and how many directory authorities supported every single decision that went into the Tor consensus, and provides the foundation for all the results presented in this section.

6.1.2 Fast Relays

While our interest in this research was triggered by the variations in the assignment of the HSDir flag, we start our analysis with the Fast flag, because it is a prerequisite for the HSDir flag and due to the three different methods of determining bandwidth, we believed it to be the most likely reason for relays temporarily losing the HSDir flag.

According to Figure 6.1, this theory appears to be incorrect, because the number of relays granted the Fast flag does not fluctuate nearly as much as the number of relays with the HSDir flag. This also matched our own experience, since our relays had retained their Fast flags during the period where they were not granted the HSDir flag.

At this point it is important to remember that obtaining the Fast flag only means that more than half of the directory authorities believed the relay to be fast. Directory authorities which do not believe a relay to be fast (or stable for that matter) will never consider granting the HSDir flag. For example, if a relay

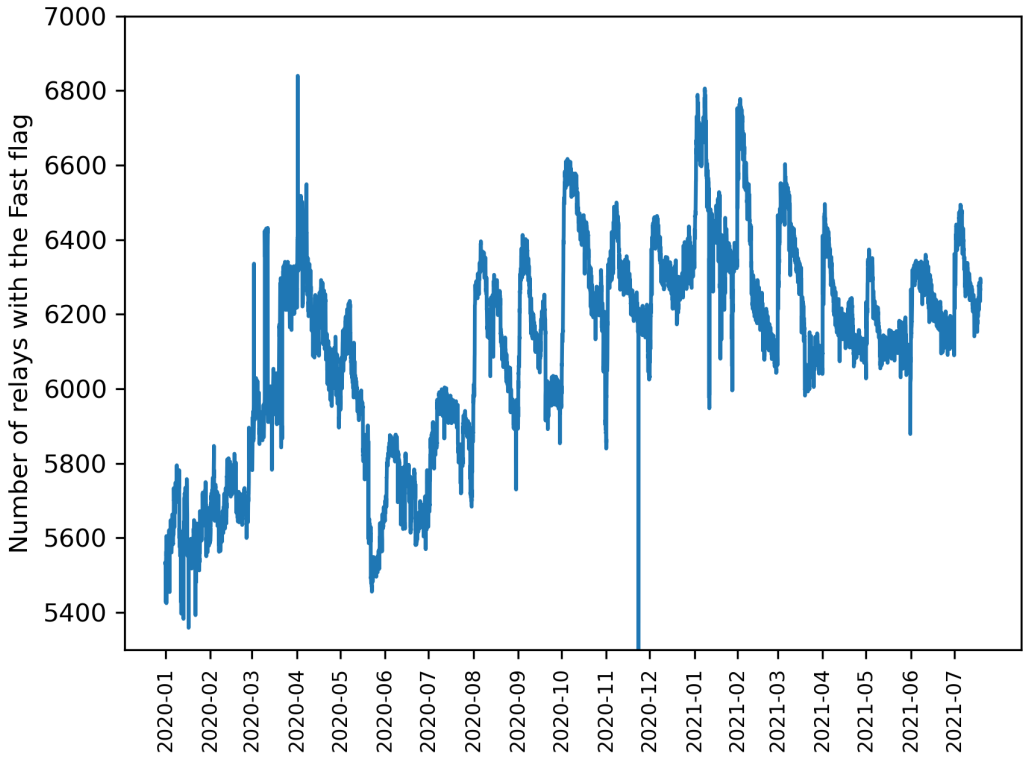

Figure 6.1: Number of Fast relays in the consensus

gets 5/9 votes for the Fast and Stable flags, it will obtain both flags, but if just a single one of those 5 authorities does not believe the relay to be up for more than 96 hours, it will not obtain the HSDir flag.

To visualize this aspect, we parsed the archived votes of all directory authorities and evaluated for every relay in the consensus how many votes for the Fast flag it received. The results of this analysis are visualized in Figure 6.2 and show that most relays received the Fast flag with 100 % of the votes. It also confirms several of the voting patterns we expected to see based on the current state of bandwidth measurement strategies. There are almost no relays that receive one or two votes for the Fast flag, but there is a noticeable chunk that receives three votes. This is caused by the directory authorities that are not bandwidth authorities, as they have to believe the data reported by the relays themselves.

At first, it seems like these are relays that advertise more bandwidth than they can actually provide. However, that is not exactly what the measurement tells us because bandwidth tests are obviously taking place in parallel with ordinary operation. If a relay is already handling one other connection during a bandwidth test, it will split the available bandwidth between both connections, meaning that the measured bandwidth is much lower than what was actually made available to the Tor network. A critical setting in this regard is the *MaxBandwidthBurst* configuration option, that tells Tor what amount of bandwidth it is allowed to consume at most. When we deployed relays with a bandwidth and bandwidth burst limit of 105 KB/s, they were never found to be Fast despite all of them fully providing their advertised bandwidth. Only after increasing the burst rate to several times the bandwidth rate, relays started to measure as Fast. It may therefore be the case that relays advertising enough bandwidth for the Fast flag do not receive this flag because they don't have the

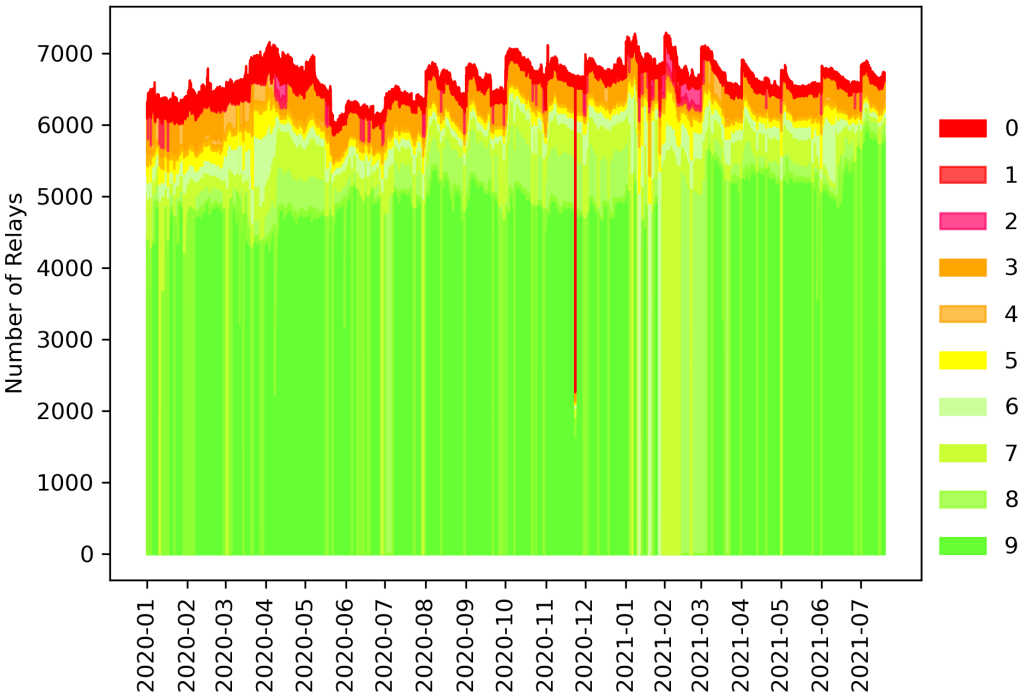

Figure 6.2: How many votes for the Fast flag did relays receive

needed burst capabilities to be measured as such.

Also worth mentioning is the fact that the relative share of relays that are considered Fast by all directory authorities is increasing, so it seems like the Tor consensus is actually getting more stable in that regard. This improvement is likely caused by the fact that internet bandwidth is increasing globally and most web applications and services expect users to have more than 105 KB/s of bandwidth available. Raising the requirements for the Fast flag might be a good idea to improve user experience when using the Tor network.

Another interesting observation we made when analyzing the voting behavior on the Fast flag was that the number of voting relays is regularly lower than nine. This includes February and March 2021, where the Tor consensus regularly contained only 7 or 8 different votes. In theory, the Tor consensus should be very resilient to the failure of individual directory authorities, so there has to be another factor causing the high volatility of the HSDir flag.

6.1.3 HSDir Relays

Figure 6.3 visualizes the number of votes for the HSDir flag received by relays in the consensus. It clearly shows that the number of relays that receive the HSDir flag from all directory authorities is less than 1000 meaning that more than 75 % of all HSDir relays in the consensus are relying on less than 9 votes. Secondly, we can clearly see a negative spike around February 2021, where the number of relays with 4 and 5 votes suddenly spikes. Since this is the time when some directory authorities stopped voting, this confirms that the non-voting directory authorities were the ones relays previously relied upon to obtain the HSDir flag.

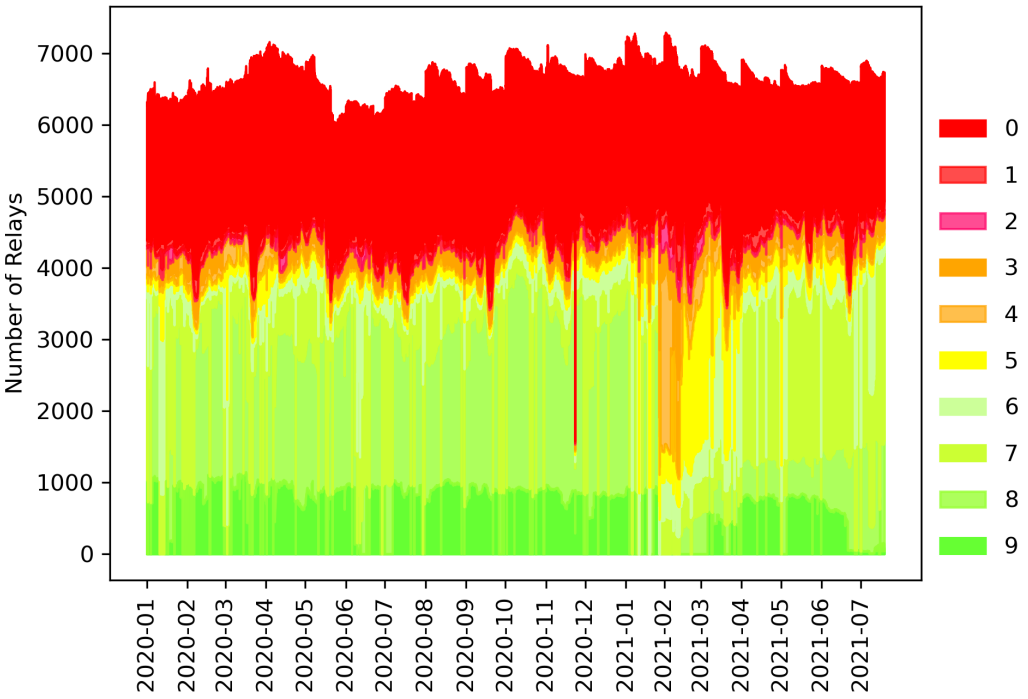

Figure 6.3: How many votes for the HSDir flag did relays receive

But the real question to ask at this point is why the level of disagreement between the votes of the directory authorities is so much larger for the HSDir flag than it is for other flags like Fast. For that purpose, we define a new metric: The number of dissenting votes. A dissenting vote happens when a directory authority granted or withheld a flag in a vote, which ended up being granted by the consensus.

Figure 6.4 shows the relative amount of dissenting votes for different flags. Unsurprisingly, it confirms that the number of dissenting votes is highest for the HSDir flag, reaching up to 25 % of the overall votes. Considering that the maximum number of possible dissenting votes is limited at 44.5 % for nine voting directory authorities, this level of dissent is reason for concern. The fact that both Fast and Stable have significantly lower levels of dissent than HSDir seems to imply that some directory authorities have trouble confirming the 96 hour uptime of relays. Uptime is tracked but not published by directory authorities, so there is no easy way to confirm this assumption. However, there is an easy way to disprove it by checking if a directory authority considered a relay Running for the last 96 hours. Checking the archived votes reveals that some directory authorities do not grant the HSDir flag to relays, even if they believe them to be Fast, Stable and Running for more than 96 hours.

Figure 6.5 shows how many dissenting votes for the HSDir flag were issued by each of the directory authorities. Note that the directory authority *moria1* has a significantly different voting behavior for this flag. While other directory authorities tend to have a very low number of dissenting votes with occasional spikes that can be explained by temporary issues when measuring uptime or bandwidth, *moria1* constantly disagrees on at least 3000 votes. This nicely aligns with the previous observation from Figure 6.3 which shows that only a small amount of relays manages to obtain 100 % of votes.

This behavior seems to be intentional, as the operator of *moria1* is one of the

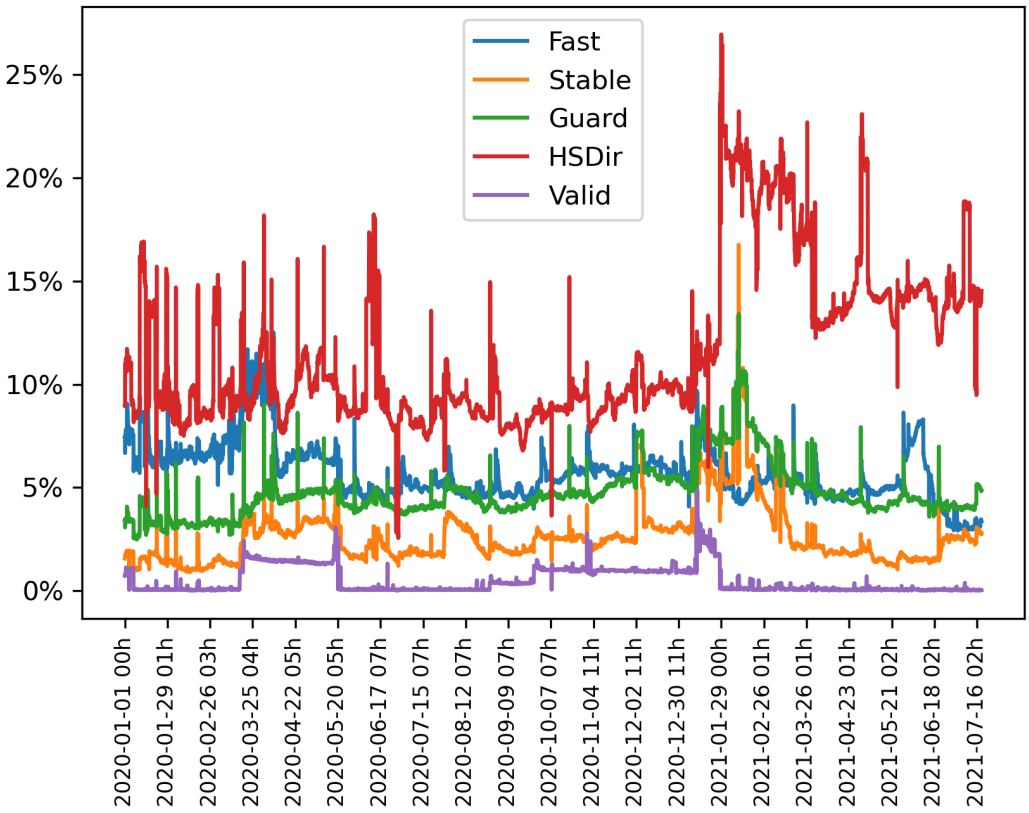

Figure 6.4: Dissenting votes per flag

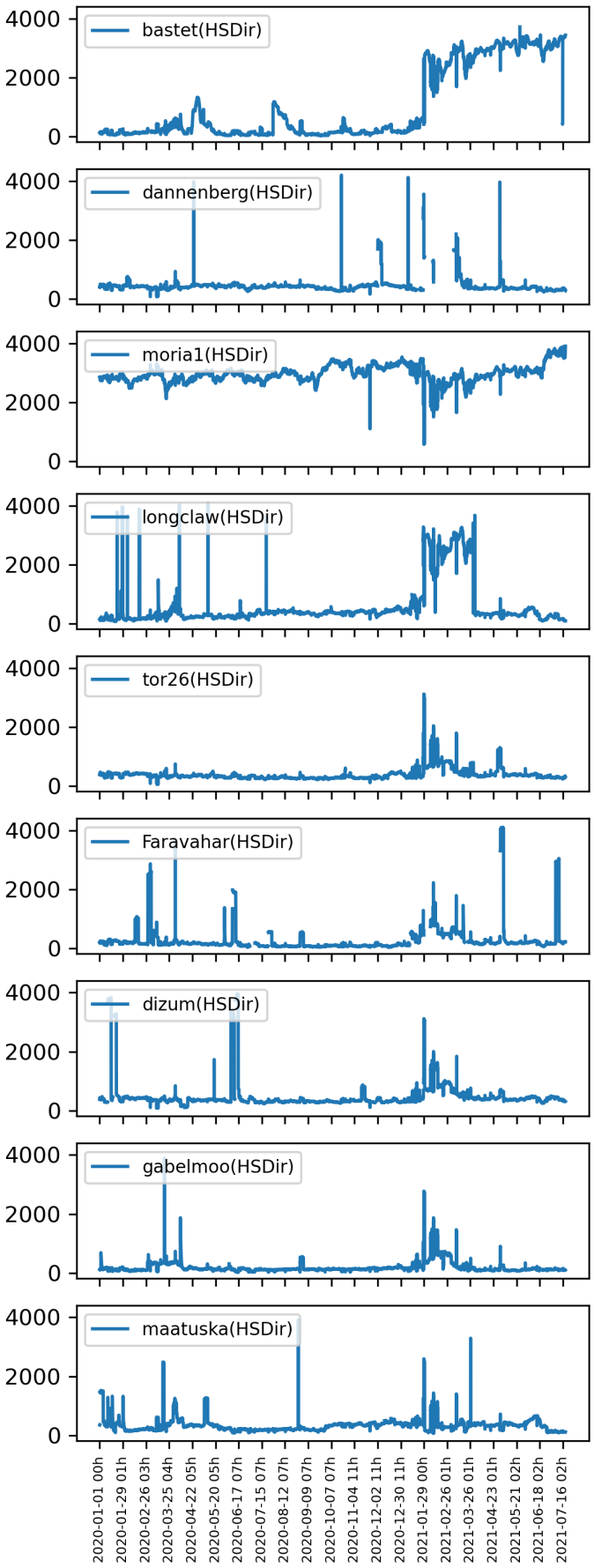

Figure 6.5: Dissenting HSDir votes per relay

original designers of the Tor network and is known to use his directory authority to test future changes and improvements to the Tor network [54]. While we do not believe that this is a very good argument as there are ways to test potential future flag requirements without actively introducing dissent in the current consensus, a single directory authority not following the directory specification does not explain the observed fluctuations in the number of HSDir relays.

To answer this question, special attention should be paid to the events around January 28th, 2021 when the directory authorities *bastet* and *longclaw* suddenly changed their voting behavior to align with moria1. According to the relay operator mailing list [28] a denial of service attack against directory authorities was detected on that day that forced several directory authorities to go offline. Based on this observation, we theorize that in response to this attack the Tor team developed a quick fix on top of the development branch that moria1 was using and made that available to other directory authorities as well to stabilize the network. That would make the changed voting behavior for HSDir flags an unintended side effect. Unfortunately, there is no way to confirm this theory because Tor relays only publish their version string without any further indication of the actual source code they are running. Both before and after January 28th, 2021 *moria1*, *bastet*, and *longclaw* published the version string *0.4.6.0-alpha-dev* indicating that their Tor binary was compiled off a development branch. The fact that these directory authorities changed their voting behavior without changing their version string clearly illustrates that current version information provides little insight into what code is actually being run by a relay. Future versions of Tor should strive to provide more transparency regarding the source code operated by directory authorities. Including the commit hash of the source code that was used to build Tor is one one proposition to achieve this.

What we have been unable to confirm is whether this dissenting voting behavior is intentional or not. Longclaw returned to the officially specified voting behavior at the end of March 2021 but *bastet* did not, although their inconsistent voting behavior was reported [41], so the Tor project must be aware of it. Interestingly, when *longclaw* reverted to voting according to the directory specification, their version string did change to *0.4.5.7*. So they moved from a development build to an older official Tor release. This leaves us wondering if two authorities voting based on different criteria than the others provides any benefits to the Tor network that justify the dissent they are causing.

Ultimately, the high fluctuations in the hidden service directory were caused by a mixture of several issues. First the changed voting behavior of three directory authorities reduced the amount of obtainable votes to six. If any of the remaining six relays went offline – which tends to happen during ongoing DoS attacks – relays needed to obtain five out of five available votes. So any individual measurement failure regarding either bandwidth or uptime led to a withdrawn HSDir flag.

6.1.4 Other voting inconsistencies

After noticing the inconsistent voting behavior for the HSDir flag, we obviously asked ourselves if the same issue also applies to other flags. For that purpose, we ran the same evaluation for all the other flags that can be assigned to relays and found two more hints towards inconsistent voting criteria.

The first again seems to be tied to *moria1* and applies to the flags Valid, Exit and V2Dir. Figure 6.6 only shows the dissent for the Valid flag because the graphs

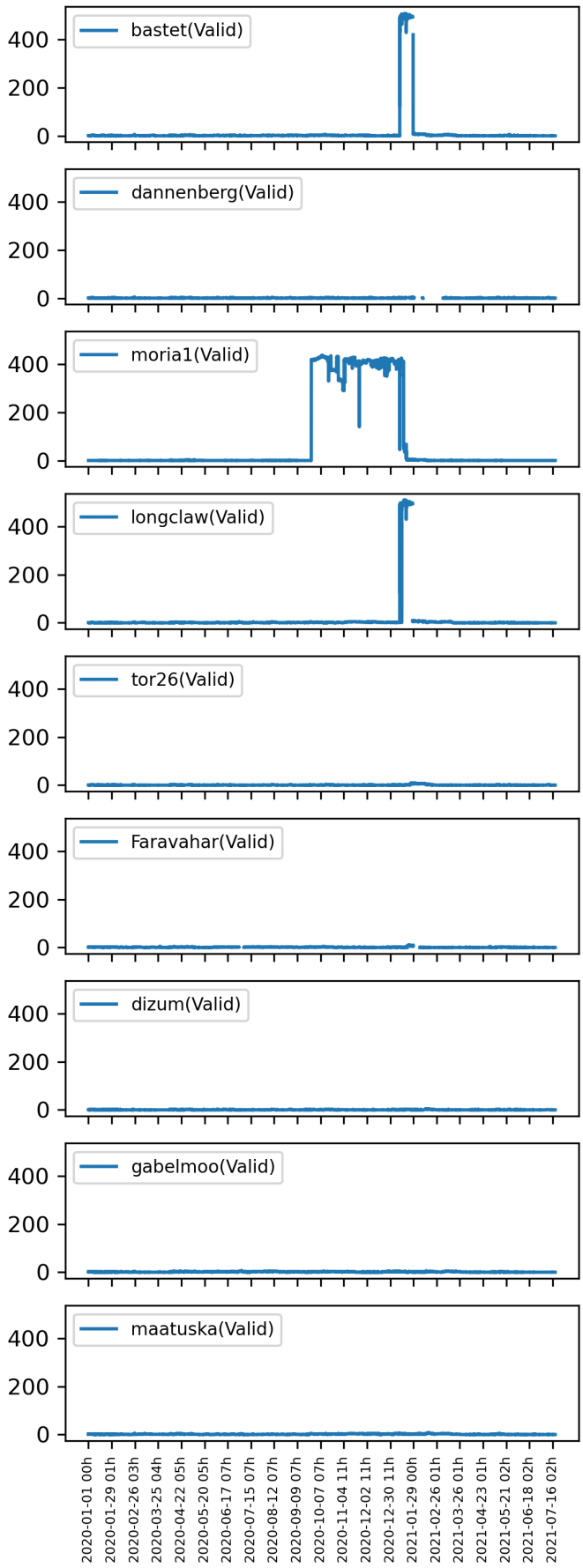

Figure 6.6: Dissenting Valid votes per relay

for the other flags look exactly the same, which leads us to believe that the dissent for all three flags was caused by a common issue. Our data shows that the dissenting votes from *moria1* started on September 26th, 2020 and continued until January 28th, 2021. The relays *bastet* and *longclaw* adopted the voting behavior from *moria1* on January 12th, 2021. At this point we have to consider the adoption of unspecified voting behavior from *moria1* by *bastet* and *longclaw* a pattern. Ironically, the change that ended this inconsistent voting pattern by all three directory authorities also caused *bastet* and *longclaw* to begin voting against specification for the HSDir flag. So in a way we traded one inconsistency for another.

The second inconsistency we found concerns the BadExit flag. This flag is a little different from the previous ones, because there are no clear requirements as to what a relay must do to earn this flag. According to the specification, this flag should be given to exits that are believed to be useless as an exit node. This would be the case if an ISP censors outgoing traffic or a firewall is too restrictive and prevents Tor users from actually reaching the resources they are interested in via this exit relay. Since there are no clear criteria defined, a group within the Tor project tries to monitor the network for bad exits and flags them as such. While they certainly are utilizing automation for this task, a non-negligible part of their work relies on Tor users reporting relays that do not work as expected.

Considering the fact that bad exits are actively monitored, we were quite surprised to see that the dissent on bad exits in Figure 6.7 also shows two clearly unique patterns. *bastet* and *dizum* hold a reproducible minority opinion regarding the BadExit flag. *Maatuska* started supporting them in September 2020, leaving us again with three directory authorities that seem to consistently vote differently from the other directory authorities. Since there are no requirements specified for the BadExit flag, we are unable to find out if this behavior is in fact caused by two different sets of criteria for the flag, if there are two different measurement methods, or if this actually valid because some relays only work from the perspective of certain directory authorities. Without internal knowledge about the workings of the Tor bad-relays team, it is impossible to investigate this phenomenon any further.

6.1.5 Monthly relay spikes

The final observation we would like to present in this work regards the composition of the Tor consensus. Attentive readers may have already noticed in Figures 6.2 and 6.3 that the total number of relays that are being voted on follows a specific pattern that spikes at a certain point in time and then decreases for a while before spiking again. Figure 6.8 highlights this behavior more clearly and shows these spikes reliably occur on the first of every month since July 2020. To be more precise they all join the Tor network within the first minute of the first day of a new month. There is no clear pattern as to when they leave, but it seems like they randomly drop out of the network over time. This behavior was independently noticed by the Tor project[1], but so far they have no explanations as to why it happens. The findings presented in this chapter were of course made available to the Tor project prior to publication.

The first question we asked was if those spikes were caused by new relays joining the network on a monthly basis or old relays rejoining. By analyzing the archived consensus information we were able to identify 90 relays that had rejoined the network at the beginning of every month since July 2020. Furthermore, we found 230 relays that contributed to at least 10 of the 13 recorded

[1]https://gitlab.torproject.org/tpo/network-health/team/-/issues/76

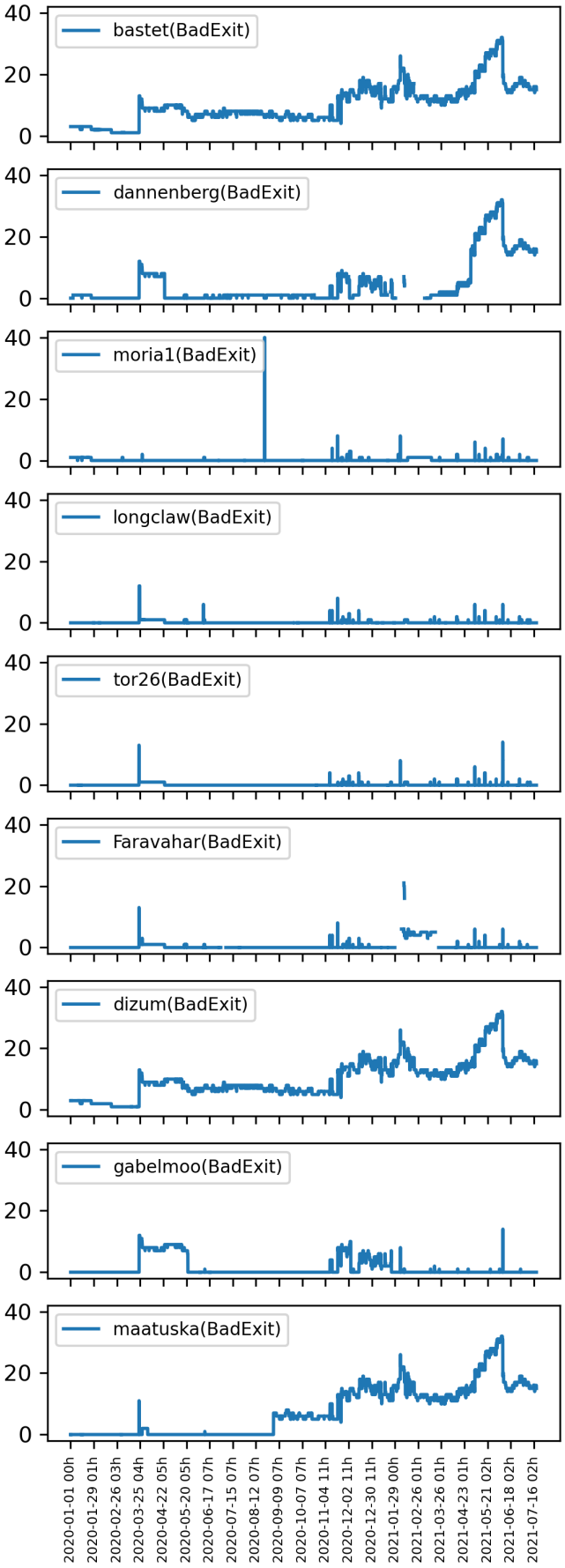

Figure 6.7: Dissenting BadExit votes per relay

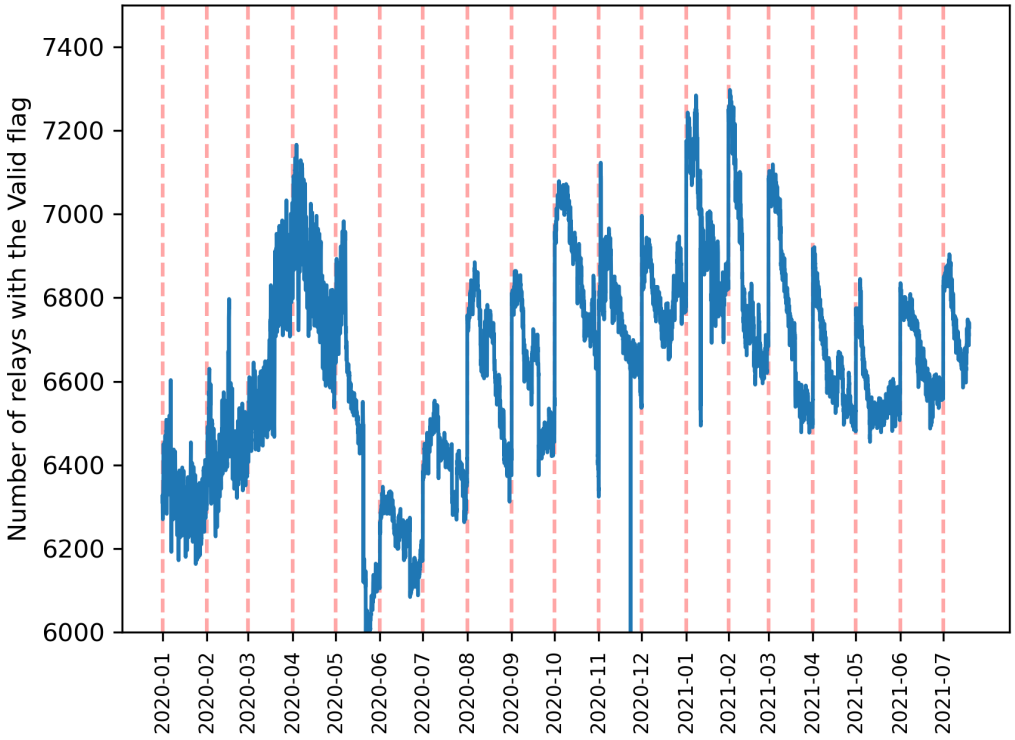

Figure 6.8: Highlight monthly spikes in number of valid relays

monthly relay spikes. So we can confirm that these relay spikes are caused by relays regularly joining and leaving the Tor network.

The regular timing supports the theory that this pattern is produced by a common misconfiguration shared by all those relays. One of Tor's configuration options enables a relay to limit the amount of bandwidth used during a given time interval (*AccountingMax*). This is very useful for users with a strict limit on how much bandwidth they are allowed to consume and could be a potential explanation. If a relay is configured to use a limited amount of bandwidth per month, the observed pattern of relays joining the network at the beginning of a new month and leaving randomly when they run out of bandwidth makes sense. The only problem with this theory is that Tor should not behave like this when this option is enabled. According to the documentation[2], a relay that runs out of bandwidth hibernates until a random time within the next time period to avoid all relays starting at the same time. Unless there is a bug affecting several Tor implementations, this theory does not explain the regular monthly spikes, but it may very well explain why these relays leave the Tor network after a random period of time.

To find out if the relays responsible for this phenomenon have anything in common that might shed light on the subject, we extracted information about them from the consensus documents and relay descriptors published at the beginning of every month. Apart from the unique fingerprint that we used to identify rejoining relays, the Tor consensus reveals the IP address and the Tor version running on the relay. Additionally, the consensus provides the digest needed to query server descriptors with more information about a relay, like its uptime, family, or contact information. Additionally, we used reverse DNS

[2]https://www.torproject.org/docs/tor-manual.html.en

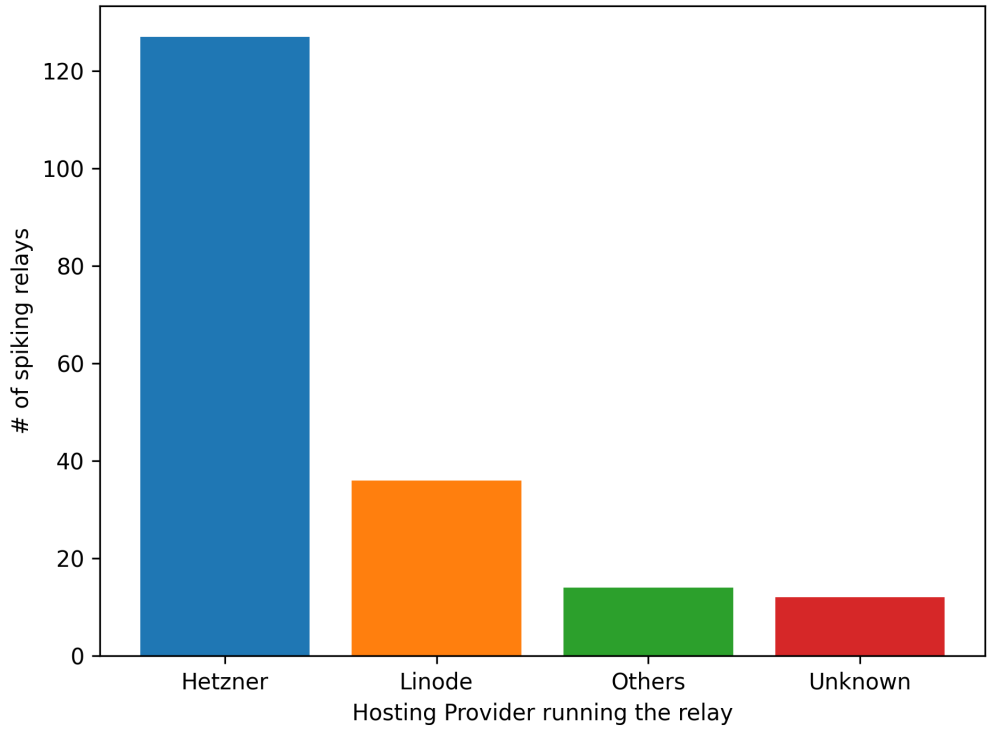

Figure 6.9: Provider assignment based on hostname for 189 relays that spiked at least 10 of 13 times

lookups to assign hostnames to the IP addresses of the relays. Unfortunately, there were no obvious commonalities between the different relays. The only thing of interest is that reverse DNS responses tie a majority of relays back to very few large cloud hosters. Figure 6.9 shows that most of the relays that spiked more than 10 times were hosted at the German Hetzner Online GmbH[3] which is one of the largest operators of Tor relays. The other hoster, Linode, LLC[4], is also responsible for a significant number of Tor relays. While it is common for relays to be run at cloud hosters, these two cloud hosters contribute to these relay spikes far more than they contribute to the overall number of relays. Other cloud hosters that are used to operate lots of relays like OVH do not show up in our data at all, so the issue seems to be related to these hosters in some way. This argument gets even stronger when we compare the total number of relays operated at those providers to the number of relays with monthly reappearances. Hetzner operates 440 relays, of which 146 (33 %) are contributing to the monthly relay spikes. For Linode the relation is even worse with 216 relays in total and 118 relays (54 %) contributing to relay spikes. This leads to the conclusion that there is either a widely distributed Tor setup that uses an accounting limit and forces a reboot of the Tor process at the beginning of every month or there is one actor running all those relays on different cloud providers who happens to have an accounting limit and a monthly reboot policy in place.

The tutorials for running Tor relays on both cloud providers [112, 131] do include an accounting limit, but say nothing about monthly reboots and we could not find any public resources that would explain a large number of users setting the same monthly reboot policy. On the other hand however, we detected a weird

[3]https://www.hetzner.com/
[4]https://www.linode.com/

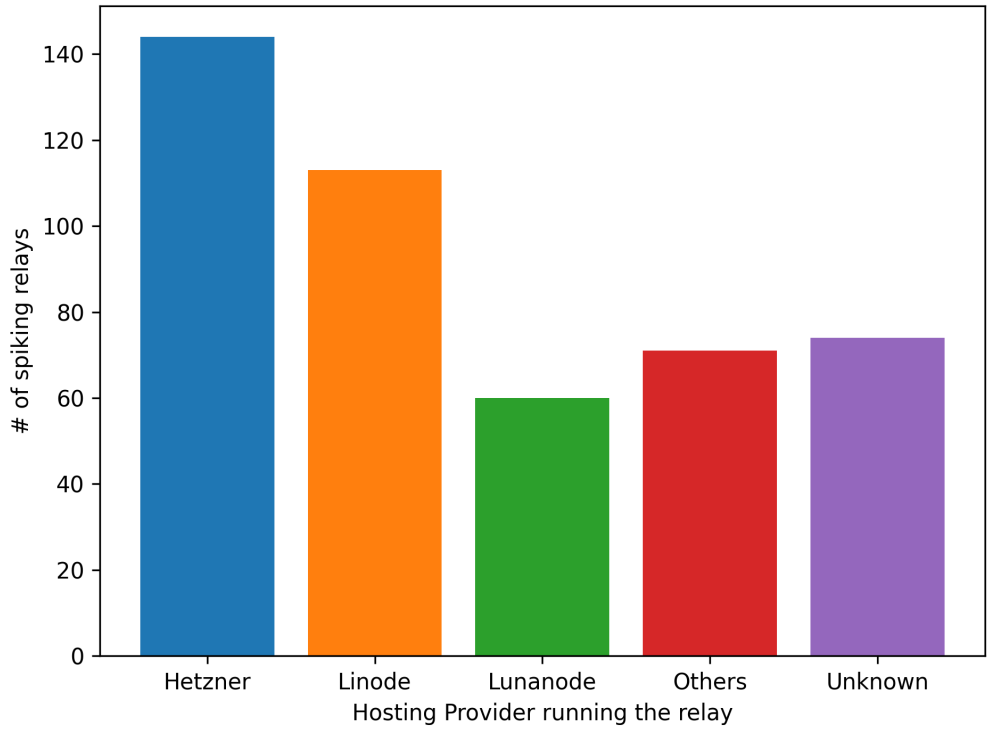

Figure 6.10: Provider assignment based on hostname for 462 relays that spiked at least 4 of 13 times

pattern in when relays that contribute monthly spikes first joined the network. A vast majority joined between April and June 2020, and they did so in ordered time intervals. For example, between April 29th and May 10th, 39 relays that contribute to relay spikes were deployed at Hetzner. Not a single one showed up at any other hoster during that period. A week later between May 18th and May 26th, 22 relays that contribute to relay spikes were deployed at Linode and during that period not a single one was deployed at Hetzner. While this is no conclusive proof, the probability of such a pattern emerging from random users deploying Tor relays seems negligible, even if there were a shared configuration source that is responsible for the monthly spikes.

The final observation we can contribute is that if we include relays that have been contributing to relay spikes at least 4 times (see Figure 6.10), a third cloud hosting provider, Lunanode[5], shows up with all relays having contributed to between 4 and 6 spikes. Combined with the fact that none of those relays are still running, this indicates that there was a third cloud hosting provider that was used in the past to operate this kind of relay. Unfortunately, we were unable to find any hints on what those relays are being used for and could therefore not tell if they are malicious. Further research by the Tor project [99] revealed them to be operated by an entity named KAX17, who is indeed linked to malicious behavior. The relays were removed from the Tor network in December 2021 [74] for being operated by the same entity. No conclusions were published on who is responsible for deploying those relays or what kind of attack they were trying to conduct, most likely because the analysis by the Tor project was limited to the same data available to us and this data just does not provide enough information to retroactively analyze how malicious relays attacked the network.

[5]https://www.lunanode.com/

6.2 Summary

Our analysis has found several inconsistencies within the Tor consensus that have the potential to negatively impact Tor users by limiting the number of relays available to them without good reason. The most important aspect to improve upon would be to increase transparency on what specification directory authorities are currently employing. Just in the last year we have encountered multiple occasions where directory authorities clearly changed their voting behavior without publicly disclosing it or at least giving some indication of a change in their Tor version string. The Tor version strings themselves turn out to be insufficient because multiple directory authorities use self-compiled developer versions of Tor, where the version string tells us almost nothing about the actually running code. A valuable improvement would be to include the commit hash and branch of the code in manually compiled Tor versions. This would still allow the Tor project to deploy hotfixes directly to directory authorities but keep transparency on when the running Tor version has changed. If the development branches are publicly visible, external analysts would even be able to find out if a deviation from the official directory specification is intended or not, which would greatly improve the transparency of the voting process. The endeavor to re-implement the Tor client in Rust also provides an opportunity to make the Tor clients build process reproducible. With reproducible builds, the version string could also include a hash value of the binary produced by the build process. This enables external observers to verify and reproduce the behavior of other Tor relays and increases the probability that malicious relays can be detected.

Furthermore, we encourage a reevaluation of absolute flag criteria like the bandwidth required to obtain the Fast flag. The modern web is constantly developing and what was considered an acceptable bandwidth ten years ago is no longer fitting today. In order for such flags to retain their usefulness, they should either drop requirements specified in absolute values or have a process in place to ensure they are updated regularly.

We also suggest searching for better ways to test potential directory specification changes. The current strategy of having a single directory authority voting differently from the others and occasionally handing those changes out to other authorities has already disrupted the Tor consensus more than once. One could either introduce a new directory authority that only creates internal vote previews without actually publishing votes or just have existing directory authorities log the data upon which they base their decisions. This would also help to avoid issues where a hotfix tested on one directory authority is accidentally bundled with voting behavior changes that were only intended for testing purposes.

Additionally, we believe that additional measures should be implemented to automatically detect suspicious spikes or patterns in the number of available relays. Malicious actors starting up a large number of Tor relays to launch attacks have been detected [100] in the past and situations where hundreds of relays join the network over a short period of time should be automatically detected by the Tor project and not go unnoticed for a year, especially if they stand out like this. Even if there is no sign that these relays are acting maliciously, the evidence hinting at those relays being run by the same entity should have been noticed by the Tor project earlier.

Finally, we would like to emphasize that our analysis has not found a single instance where we believe that malicious actors successfully modified the Tor consensus to attack users. Considering that Tor is a target for several nation state actors [8], we have to assume that attacks on the Tor consensus would

have been launched if they were easy to execute. Despite the issues and improvement suggestions mentioned in this paper, it appears that Tor's method of forming a secure consensus in a globally distributed network does indeed withstand the test of time.

Chapter 7

Improving Tor onion services

The research within the previous chapters has indeed confirmed that the Tor network should be capable of sustaining a distributed digital identity network. However, it also identifies several points where realistic real-world improvements can be achieved by minor modifications to the Tor client. This chapter presents three potential changes that can be implemented on individual Tor clients without requiring additional support from the Tor network that would significantly improve the performance of Tor-aware distributed applications in general and the Digidow project's networking scheme in particular. These improvements affect the load put on the network by an onion service, the wait times experienced by users, and the power consumption required for networking.

7.1 Using Current Onion Services

The networking approach presented in section 3 can be implemented with Tor's current onion service implementation. In order to have a baseline for comparison, this section provides an outline of how networking, especially concepts like Pub/Sub would work with current onion services. To highlight the inefficiencies of such an implementation, we propose three metrics:

- First, the number of Tor circuits needed to either register to or notify about a match. This metric captures the load put on the Tor network as every circuit requires communication between Tor nodes.
- Second, the number of Tor circuits that need to be created or cannibalized. This distinction is important because some circuits do not have a defined destination. E.g., the rendezvous point can be any Tor node chosen by the client, allowing the client to just use one of the general circuits that should be available. Other circuits need to end at a specific relay that is not known in advance, forcing Tor to wait until a circuit has either been constructed or successfully cannibalized, directly adding to the overall time needed to establish communication via an onion service.
- The third metric captures the number of sequential Tor circuits that communication has to go through. Every circuit consists of three Tor nodes and each of those nodes adds network delay to a transferred message. A concept like Tor onion services that communicates via several Tor circuits in a row accrues network latency, which is very likely to negatively impact user experience.

Figure 7.1 illustrates the number of circuits needed to set up a callback onion service and to subscribe it to an event. Some circuits have been assigned the same number to indicate that those circuits are created in parallel, while sequential numbers indicate that these circuits can only be created after the previous circuits have been established and used.

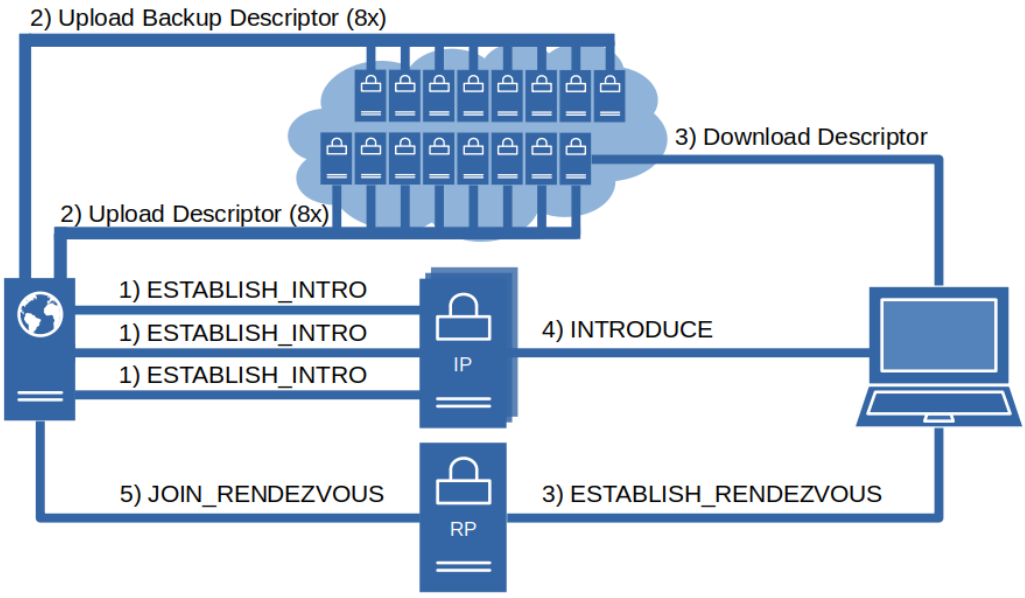

Figure 7.1: Number of circuits created for an onion service

Creating and publishing a new onion service requires 19 circuits in two stages. In the first stage, three circuits to introduction points are created, usually by cannibalizing general circuits. The second stage requires 16 circuits to specific Tor nodes within the hidden service directory. This is more than Tor usually keeps prepared, forcing Tor to wait for new circuits to be created.

Connecting to an onion service requires only 4 circuits, but those circuits are created over 3 different stages. The first stage establishes circuits to a specific node within the HSDIR and an arbitrary node as *RP*. The second stage creates a circuit to the *IP* specified in the descriptor loaded from the HSDIR and the final stage is triggered when the onion service receives the forwarded introduction request from the *IP* and establishes a circuit to the *RP*.

Figure 7.2 illustrates the sequence of messages that must all be exchanged via circuits before the connection to an onion service can be established. Since we are mainly interested in the network latency added by circuits, multiple circuits being used in parallel are not shown here. Creating an onion service only requires three message exchanges via circuits, while connecting to an onion service requires six message exchanges. Keeping in mind that every Tor circuit consists of three nodes, that adds the accrued latency of 18 Tor nodes before communication with an onion service is possible. At this point it should be stressed again that no content data has been exchanged between client and onion service at this point, as this is just the effort required to allow exchanging data.

One could argue that it is unfair to include the time needed to set up an onion service in our analysis, as this is not necessary to connect to an onion service. In a Pub/Sub scenario via Tor however, subscribers have to create their own callback service descriptors. It it reasonable to assume that subscribers would only set up this callback onion service after they have decided to subscribe to an event for the first time. This directly adds the onion service creation time to the absolute time needed to subscribe to an event. Further subscription later on could of course re-use the already existing onion service.

Another reason to include the time needed to set up an onion service in our

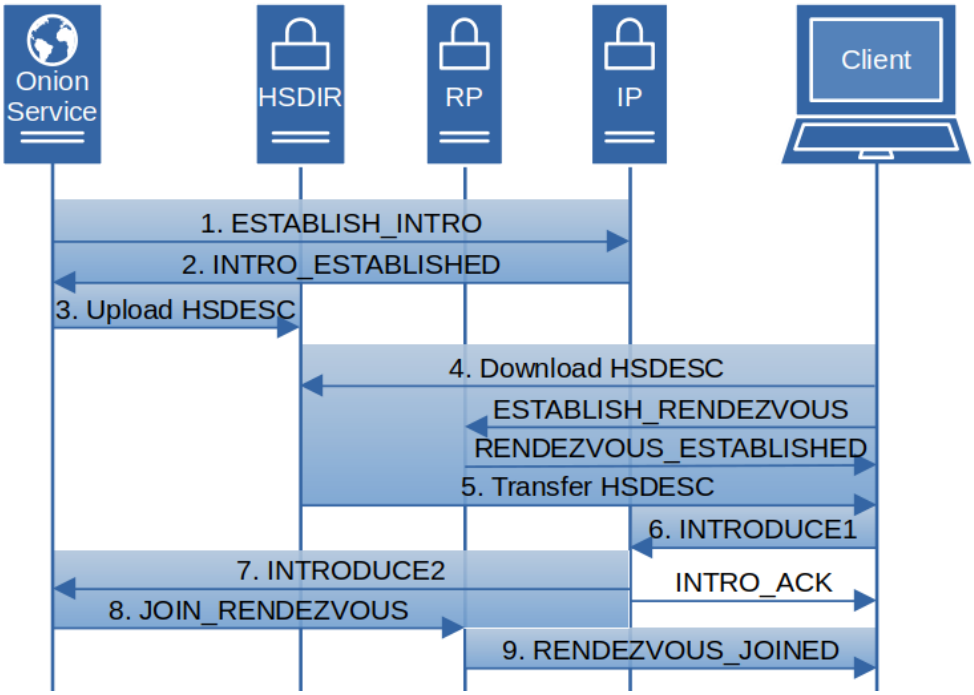

Figure 7.2: Circuit latency added before connecting to an onion service

analysis is that persistent callback onion services, when used over a long time period, can themselves become linkable identifiers for user tracking from the point of view of publishers. Therefore, we expect subscribers to regularly rotate their callback services (to the point of making them single-use) to provide unlinkability on this level, requiring the regular creation of onion services.

The process of notifying a subscriber about an event requires an established rendezvous circuit between publisher and subscriber, but it does not need to create a new onion service. So it incurs the same setup penalties as a subscription process that does not need to create an onion service. The significant change is that a publisher will usually have to notify multiple subscribers at once, forcing a single Tor client to create simultaneous connections to different onion services with every one of them needing 3 different circuits, which makes it likely that the Tor client will run out of prepared circuits and end up having to wait for new circuits to be created.

To give a concrete idea of just how much time is needed for Pub/Sub communication via onion services, we implemented a simple prototype that deploys two onion services, has one service subscribe for an event at the other one, and receives a notification about an event happening. The results, which are presented in detail in section 7.5, show that Tor added on average 19 seconds to a single Pub/Sub exchange, making it quite unfeasible for many modern applications—especially interactive services like instant messaging or digital identity use.

7.2 Improvement: Make the HSDIR Optional

The first component of onion services that warrants investigation is the hidden service directory. Publishing service descriptors requires multiple circuits ev-

ery time, putting significant load upon the Tor network because clients always have to communicate with the hidden service directory before they can start connecting to an introduction point. From a functional perspective, the hidden service directory is responsible for translating onion addresses to current service descriptors. Unlike onion addresses, service descriptors need to be renewed regularly because introduction points can go offline, forcing onion services to create new introduction points and update their service descriptors. In terms of privacy, the hidden service directory has been proven to be a point of potential attacks in the past [11, 102], forcing Tor to continuously improve the privacy protections on the hidden service directory. Chapter 4 has shown that the hidden service directory is still leaking information about both users and operators of onion services and chapter 5 has found that publishing to the onion service is the most time consuming step of the onion service creation process.

The main idea behind the first suggested improvement is simple: Distributed systems already need a mechanism to exchange contact information about all members. In the Digidow architecture, the sensor directory would serve that purpose by contact information (traditionally either hostnames or IP addresses) to PIAs. If this contact information happens to be onion addresses, members of a distributed system need to first obtain the onion addresses and then translate those onion addresses to service descriptors, before they can actually start a connection. Tor-aware distributed systems could remove one indirection step by directly using service descriptors as contact information.

This introduces one new problem, namely that that the contact information must be updated more regularly. Thankfully, the current Tor implementation already assigns a validity period to service descriptors, which can be used to implement different approaches for this problem: Some applications might choose to just create new onion services instead of updating their old ones, allowing the old descriptors to expire (with the added benefit of better unlinkability of callback services). If clients already need to set up new introduction points, there is little difference between the effort of setting up a fresh onion service or keeping around the old one. If the publisher also does not need to recognize clients, like it is the case when registering at a sensor, this problem can just be ignored. Other clients that use onion services for instant messaging for example, would probably have to actively inform every contact about changes to their descriptor. In return, their messages would be transmitted more quickly and demand less resources from the Tor network.

Therefore, we propose an extension of the Tor protocol to optionally allow the creation of onion services that are not published to the hidden service directory. The service descriptor should just be made available locally, with the responsibility for further distribution delegated to the application. In the same way, Tor should also notify a managing application about changes made to a service descriptor because of changing introduction points. Finally, other Tor clients need a way to load a service descriptor from an external source that is not the hidden service directory.

7.2.1 Implementation

Our proposal has already identified the three changes that are necessary to realize onion services without using the hidden service directory. Thankfully, the first step—not uploading service descriptors to the hidden service directory—is already implemented as a configuration option (*PublishHidServDescriptors*). The Tor man page explains that this option is only useful if there exists a

Tor controller that handles descriptor publication. Unfortunately, Tor controllers have no way to either learn unpublished service descriptors or load received service descriptors, rendering this feature mostly useless. Most likely, the missing functionality was intended to be added to Tor, so our prototype only adds two features that were at least at some point meant to be a part of Tor.

As discussed in section 2.4, there are two potential ways for applications to communicate with Tor. One can either use Tor as a SOCKS proxy or use the *control protocol* [122] to have full control over Tor's behavior. We expect Tor-aware applications will always end up using both. The control protocol to configure Tor according to their needs and SOCKS to exchange data with other clients.

The control protocol already supports subscribing to events (another example of Pub/Sub being used for communication between applications) and there exists an event for *HS_DESC_CONTENT*. However, this event is only triggered if a descriptor is downloaded from the hidden service directory, not if a new descriptor is created. Our prototype modifies Tor to also publish this event whenever a new onion service is created without being published to the HSDIR. Finally, we need a way to load a service descriptor directly. The current implementation does not provide that functionality yet, but it does provide a similar one. The Tor network uses other descriptors to describe every node within the network and the control protocol can already load such node descriptors dynamically with the *POSTDESCRIPTOR* function. Following the same idea, we implemented a new *POSTHSDESCRIPTOR* function in the control protocol that supports loading hidden service descriptors directly. Since this is a new function within the control protocol, it is a non-breaking change that does not impact any of the current functionality of Tor.

7.2.2 Limitations

Generally, this approach is only viable if the circumstances provide a meaningful way of exchanging service descriptors directly. It is important to keep in mind that service descriptors are not static, so the exchange mechanism for service descriptors must be able to keep service descriptors up to date.

Our prototype implementation adds another issue because it loads fully encrypted service descriptors via the control protocol. This allows us to reuse the already existing code path to parse and store service descriptors. Unfortunately, this requires Tor to decrypt the service descriptors and decryption is dependent on the current time period, which causes our implementation to fail when time periods change after a descriptor has been created but before it has been loaded. This could be addressed by changing the implementation to both emit and load plain text service descriptors instead of encrypted ones because the descriptor plain text contains no references to the current time period anymore.

Finally, our decision to re-use an already existing event (*HS_DESC_CONTENT*) to learn about unpublished onion service descriptors might cause issues with existing Tor controllers that expect this event to only happen after a descriptor has been downloaded. We tried to minimize the number of potentially affected users by only triggering the new events, if the *PublishHidServDescriptors* option is set. If this is not sufficient, specifying a new control protocol event specifically for descriptors not uploaded to the HSDIR would completely remove this issue.

7.2.3 Privacy Analysis

Utilizing this modification removes the need to interact with the hidden service directory, meaning that the hidden service directory cannot learn any information about the client or the onion service, which constitutes a privacy improvement. From the perspective of the introduction point and the rendezvous point, the changes are transparent, because their part of the protocol remains unchanged.

The only node that could at least detect that this modification is being used is the guard relay of the server. If a Tor instance has three long lived outgoing circuits that are only used to receive information, there is a high probability that this instance is running an onion service. If that same instance does not regularly create outgoing circuits to update the service descriptor published to the hidden service directory, a guard relay could detect onion services operated with this modification. This is especially effective if the Tor instance is not creating any unrelated outgoing circuits that might have been used for uploads. Consequently, this privacy leak could be addressed by creating dummy upload circuits, but we believe that the minimal gain in privacy is not worth the additional load on the Tor network and chose not to implement this for our prototype.

It could be argued that a privacy problem is added by privacy protection tasks being moved from the hidden service directory to the unspecified contact information distribution method of a distributed system. To understand why this is not an issue, it is important to keep in mind that both onion addresses and service descriptors uniquely identify an onion service and onion addresses can easily be translated into service descriptors. Therefore, the distribution of onion addresses requires the same amount of privacy as the distribution of service descriptors, independent of the adoption of this improvement.

7.2.4 Security Analysis

Our modifications only add functionality to the control protocol, so from a security perspective the only potential risk would come from malicious actors with access to the control protocol using our new features. The obvious new risk introduced by our changes is that the capability to load arbitrary descriptors enables attackers to load manipulated service descriptors into Tor. Fortunately, service descriptors are already signed by their creator to prevent such manipulations (after all, the nodes of the hidden service directory are not trusted either). This means that the new attack surface introduced by our modification is already mitigated by the standard Tor design.

An alternative malicious action would be to overload the Tor client by loading too many service descriptors. However, access to the control protocol already allows shutting down a Tor instance, so our modification does not increase the attack surface there. The additional event containing information about created service descriptors does not grant more agency to a malicious Tor controller, but it provides new information that was previously unavailable. With providing that information to Tor controllers being the primary motivation for adding this functionality, we do not consider this a relevant security impact.

7.3 Improvement: Bundle Information in the INTRO Cell

Another aspect of onion services that could be improved is the need to use rendezvous points. The time it takes to set up a circuit via a rendezvous point is one of the main contributors to overall latency when communicating with onion services. Without them, all traffic between onion services and clients would have to run via the introduction points, making them centralized points that could be either abused to monitor onion service usage or overwhelmed by the load of popular onion services.

In the Pub/Sub model, however, communication is uni-directional: there is no need for a bidirectional rendezvous circuit, as long as we can transfer the necessary information. In order to make traffic analysis harder, Tor pads all of its cells to a standard size, meaning that a regular *INTRODUCE1* utilizes only 310 out of 498 available bytes, leaving 188 bytes of additional payload that could be added. While that is not a lot of data, it should be enough for a simple subscribe request containing only a list of event types and an onion address as callback. Even message transfer for scenarios like instant messaging is plausible. SMS for example has always had a limit of 160 characters and until 2018 Twitter only allowed Tweets with up to 140 characters.

Data transmitted in this way would reach the onion service at a time when traditional onion services only start establishing their rendezvous circuits. This saves the time it takes to create the rendezvous circuit, the time needed to link the two rendezvous circuits together, and finally the time required to actually send the message via the extended 6-hop circuit. As a side effect, this reduces the overall load on the Tor network by removing the need for both rendezvous circuits.

This modification should be completely opaque for outside observers, as they see the same number of encrypted cells going from the client to the onion service. Attackers with direct access to the onion service network traffic could notice that no outgoing rendezvous circuits are being built, which will be discussed in section 7.3.3.

So, for our second improvement, we propose to give Tor clients the option to extend their introduction requests with additional information for the onion service. This change must be opaque for the introduction point, so that our prototype remains functional on the current Tor network and does not stand out among other unmodified onion services.

7.3.1 Implementation

The implementation of this improvement is more complex than the previous one, because it has to change the contents of well-defined Tor cells. Thankfully, the Tor protocols have been designed with extensibility in mind. The onion service specification [122] defines a format for future extensions that can be used in both *INTRODUCE1* and *INTRODUCE2* cells to communicate additional information to the introduction point or the onion service. At the time of writing, there was not a single extension published that added information to any of those cells, but the parsing and encoding of extensions is already implemented, meaning that any valid extension data added on the client side, is easily available at the onion service.

That leaves three specific tasks open for the prototype implementation: First, the Tor client needs a way to send *INTRODUCE1* cells to introduction points directly. This is a new requirement, because until now they were only sent when

a client wanted to connect to an onion address via Tor's SOCKS proxy. In our new scenario, there is no SOCKS connection, because there is no communication planned via the rendezvous circuits. While the control protocol already supports modifying running circuits, it does not provide a way to launch a new circuit manually. It would be possible to implement manual cannibalization by changing a general circuit into a specific introduction circuit, but we opted for the simpler solution of adding a new control protocol function that allows us to create new introduction circuits directly. Additionally, we added a second feature that allows us to send *INTRODUCE1* cells via any circuit. Combining these two new features solves the first task, as it allows us to send introduction requests to arbitrary Tor nodes.

For the second task, the Tor client does need a way to add extension information to an *INTRODUCE1* cell. Since we already have a new control protocol feature to manually send introduction requests, it is easy to provide the extension data there as well. The tricky bit is passing the extension information on to Tor's internal functions, because the current Tor implementation does not expect extensions to be set dynamically. Instead, it assumes that extensions are static and declared in a configuration file. This means that they can be read at any point in the code by simply checking the current configuration. However, this does not work for a Pub/Sub scheme where clients might want to subscribe to different events. We ended up having to extend the method signature of multiple functions within Tor to be able to forward dynamic extension information via Tor.

Finally, the onion service needs to check incoming *INTRODUCE2* cells for extensions and notify the application managing the onion service about any incoming messages. Tor's source code already contains a comment at the position where checks for supported extensions are supposed to be implemented. All that had to be done was to implement a check for the new event data and a new control protocol event, which can be used to notify the onion service about an introduction request with an extension that provides all the necessary subscription information.

7.3.2 Limitations

An important drawback of our prototype implementation is the fact that clients embedding data in their *INTRODUCE1* cell cannot continue the data exchange via a rendezvous circuit later on, because there is no SOCKS connection that could be attached to a rendezvous circuit. Therefore, clients can only use this way of exchanging data if their entire message can be added to the *INTRODUCE1* cell. This is not a limitation of the Tor protocol, it is just a limitation of our implementation, so this could be changed in the future.

A minor performance limitation of our prototype is that it always creates a new circuit for the introduction request, instead of trying to cannibalize a circuit first. While this will slightly increase the average time needed to prepare the circuit, it can be ignored because publishers are expected to create multiple introduction circuits in parallel and will regularly run out of prepared circuits to cannibalize.

Another disadvantage would be that clients do not receive confirmation that their message has actually reached its destination. They do receive an *INTRO_ACK* cell from the introduction point letting them know if their cell has been forwarded, but any failures that occur afterwards—like the onion service failing to decrypt the message—go unnoticed. In the current implementation

this is not really a problem, as Tor always notices when the onion service does not connect to the rendezvous point in time.

Finally, it should be mentioned that this utilization of Tor extensions assumes that no other Tor extensions are being used in parallel. With every other extension that is being used, the amount of data that can be included in the *INTRODUCE1* cell decreases and we would risk to exceed the available space with our extension. However, since there has not been a single Tor extension specified so far, we consider this a tolerable issue.

7.3.3 Privacy Analysis

Since the size of the *INTRODUCE1* cell is always padded, the introduction point cannot tell the difference between cells with or without additional extensions. Again, the only node that can observe different behavior is the guard node of the onion service, as it can see long-lived circuits receiving data, which usually leads to new circuits being connected to the rendezvous point. If those connections to rendezvous points are not happening, the guard node can speculate that the onion service is most likely using such an extension.

7.3.4 Security Analysis

This improvement adds one potential attack vectors to the Tor client, namely the capabilities to send *INTRODUCE1* cells with arbitrary extension on a random circuit. Our added feature to launch circuits can be ignored, because the already existing *EXTENDCIRCUIT* command could be used to achieve the same thing, it would just be more tedious. The arguments from 7.2.4 also apply to newly added control protocol events, so they will not be repeated here. Extensions added to the introduction request are only parsed at the onion service, so this attack vector can only threaten them. The most significant risk we see here, is that insufficient validation while parsing the extension data received from the clients might lead to unexpected behavior. Since the processing of received data is application specific, we consider it out of scope for this research.

7.4 Improvement: Use Minimized Descriptors

Both improvements implemented so far are a benefit for the performance of Pub/Sub communication, but they are not yet compatible with each other. If a client wants to bundle a subscription request with an *INTRODUCE1* cell, the callback information has to fit within less than 189 bytes. For comparison, our test service descriptor with three introduction points had a size of 13655 bytes, forcing us to include the callback information as an onion address.

This raises the question of why Tor needs so much data to encode information about the three introduction points of an onion service. The specification of Tor's service descriptor format [122] includes several decisions that compound to create this issue. First, service descriptors are encrypted to prevent the HSDIR from learning anything about them. Before encryption, the plaintext is padded with null-bytes to the nearest multiple of 10K bytes. After decryption, the size of the service descriptor reduces to only 4370 bytes, telling us that the largest part of any descriptor consists of encrypted null-bytes. Next, there is a rarely used feature for onion services known as client authorization. It allows

onion services to add a second layer of encryption to their service descriptors, that only selected clients can decrypt. To prevent observers from finding out if a descriptor uses client authorization, every service descriptor contains this second encryption layer and, if it is not used, the list of clients is just filled with 16 randomly generated clients with the key for opening the second encryption layer being included in plaintext.

Dropping the irrelevant client information reduces the descriptor size further down to 2328 bytes. Finally, this contains the information about the three introduction points (770 bytes each) along with some metadata. As explained in section 2.5.1, the descriptor needs to at least include the identity of the introduction point, along with the *INTRO_AUTH_KEY* and the *ENC_KEY*. In practice, they do contain a lot of additional information: First, the identity of any Tor relay is described by a list of *link specifiers*. Every entry in that list uses another way of identifying a Tor relay and Tor requires all clients to add every link specifier they know of in that list. This means that the identity of the introduction point is specified redundantly by its IPv4 address, IPv6 address, RSA public key, and ED25519 public key, which ensures compatibility with any currently deployed Tor versions. Second, the public key of the Tor node is included, although it could also be obtained from the consensus, since it is always published there. Third, certificates are included to cross-certify that both the *INTRO_AUTH_KEY* and *ENC_KEY* were chosen by the owner of the onion service's *MASTER_KEY*. Finally, there are legacy options for onion services interacting with older Tor versions, which we will ignore because they are no longer allowed to join the Tor network [52, 73]. Overall, this results in Tor allocating 770 bytes of data for every introduction point, while only 96 bytes end up actually being required to connect to an onion service.

One obvious conclusion from this analysis is that much of the reason why descriptors are so large is tied to the fact that the HSDIR cannot be trusted. Obviously, the same applies to any other system where descriptors are being published, although some of them may have more relaxed privacy requirements. However, in a Pub/Sub scenario where a client wants to subscribe by sending a descriptor directly to an onion service without ever publishing it, those protections can be dropped because the descriptor is exchanged directly between the two parties who need to be able to read it. A minimal descriptor for an onion service containing just the 96 bytes needed to establish a connection could feasibly be included in an *INTRODUCE1* cell, enabling the combination of both previous optimizations.

7.4.1 Implementation

Implementing this feature requires a few small changes to the client control protocol to include the necessary data and a major change to the onion service side, as it needs to assemble a valid service descriptor from the received information and include that service descriptor in its internal database.

Our implementation approach adds a new optional service descriptor argument to the control protocol function for sending an *INTRODUCE1* request. If it is set, the provided service descriptor is decrypted and a random introduction point gets selected. From the chosen introduction point, information about the *INTRO_AUTH_KEY*, the *ENC_KEY*, and the identity of the *IP* (170 bytes) are extracted. At this point, our implementation optimizes memory usage by repurposing fields that are always present in an *INTRODUCE1* cell, like the identity of the *RP*. As our modified cell can never lead to a valid rendezvous, that field can be used to store the identity of the *IP* of our descriptor. This leaves only two keys which are currently added in two new extension fields.

Aside from the strictly necessary fields, our implementation supports adding two more extension values: First, the subcredential of the service is also sent, because it is needed for encrypting the content of the *INTRODUCE1* cell that should not be readable to the introduction point. However, since this subcredential is derived from the onion address, which is already included in our subscription request, we would expect this to change in more optimized implementations. Second, an option to specify a timeout was added. Regular service descriptors have a lifetime of at most 48 hours because their blinded public keys and subcredentials do change with every time period. However, clients might want to limit the duration of their subscription even further. To support this scenario, we added an optional extension value that sets the validity period of the descriptor that will be created by the Tor client receiving this information.

Once an onion service receives an introduction request with all this information, it has to construct a valid service descriptor from it that is compatible with normal service descriptors. Naturally, the minimized descriptor leaves most of the information that would usually be included empty. This works because all the validation of a service descriptor (verifying signatures, cross-checking certificates) happens before the descriptor is actually loaded into internal storage. Our implementation bypasses all of those validation steps by directly adding a descriptor to this Tor internal storage.

7.4.2 Limitations

The minimal descriptor format consumes most of the available space in the *INTRODUCE1* cell: 32 bytes for the *ENC_KEY*, 32 bytes for the *INTRO_AUTH_KEY*, 32 bytes for the subcredential, 56 bytes for the onion address, and 15 bytes for the timestamp leave only 21 bytes for the subscribe condition. However, this limitation only applies to our prototype, more efficient implementations could improve the amount of space available for the subscribe condition. Potential improvements are presented in section 8.2.8.

7.4.3 Privacy Analysis

From an external perspective, this improvement is very similar to the one presented in section 7.3. It just bundles very specific information in the *INTRODUCE1* cell that can be used to contact the subscriber later on. Therefore, the reasoning from section 7.3.3 also applies here, meaning that the guard relay of an onion service can find out if this prototype is being used. It might even be possible to learn if service descriptors are being exchanged, because connecting to an onion service, for which the descriptor is already known, always requires two circuits (one to the rendezvous point and one to the introduction point). If a Tor instance only receives information through introduction circuits and uses other circuits always in pairs of two, the guard node could reasonably assume that the onion service is using this mode of operation.

7.4.4 Security Analysis

Accepting minimized descriptors opens up a new attack vector for malicious actors, because without cross-certification and signatures, there is no way to confirm that a minimal descriptor was created by the owner of the onion service. Instead, everyone with knowledge of the original service descriptor can generate different minimal descriptors from it. As an example, attackers could

abuse this to distribute minimal descriptors that seem like they point to a well-known onion service like Facebook, while actually pointing to an onion service under their control that simply forwards the traffic, opening up new man-in-the-middle attacks.

Alternatively, attackers could abuse this by creating minimal descriptors that point to a well-known onion service like Facebook and subscribe it to every event they can find, resulting in a distributed denial-of-service attack against the onion service. Similarly, a single publisher could be overwhelmed by subscriptions from thousands of different onion services, forcing the publisher to notify them all whenever an event happens. Technically, those problems would also arise in a Pub/Sub implementation with regular onion services, but if the length of a subscription request is not limited to 188 bytes, these problems can simply be solved by signing a subscription request with the *MASTER_KEY* of the callback onion service. Solving them when using this improvement would require optimizing our implementation to support another 64 bytes of payload that can be used to carry an ED25519 signature. Ideas on how the available payload size could be increased are presented in section 8.2.8.

7.5 Performance Evaluation

In order to properly assess the improvements presented in the previous sections, it is necessary to evaluate their performance impact on the Tor network. This quantifies the benefits that are already achieved by the presented prototype implementation and provides a foundation to discuss further potential improvements.

7.5.1 Experiment Setup

The presented prototype remains compatible with the Tor relays already deployed in the Tor network. For our experiment we only need to deploy two new Tor clients which take over the roles of publisher and subscriber. With latency and circuit build times being highly dependent on the specific relays chosen by Tor to build circuits with, our experiment has to be repeated often to produce a reliable average for the current Tor network.

In order to compare our prototype against the current Tor implementation, we implemented a simple Pub/Sub communication with and without our proposed improvements and measured them both. Our prototype branched out from Tor at Git commit 8ead53330c73e9bc1b82f6b7fc8946d629063842[1], so a Tor binary built from this version serves as our baseline, which will be compared against our implemented prototype. Every run was conducted in a fresh Docker container to prevent previous runs from impacting future measurements. The host machine was deployed within our university network without restrictions to Internet access and with publicly routable IPv4 addresses. All experiments were conducted sequentially to avoid negative impacts from running more than two Tor instances at a time, and we did not observe any immediate hardware or network bottlenecks that might have impacted the measurement.

In terms of methodology, we used the methodology as Loesing et al. [87], which already inspired the experiment design in chapter 5. Just like them, we measure the time the various stages of onion service communication take by observing log events emitted via Tor's control protocol. Unlike them, we did not to run

[1]https://gitlab.torproject.org/tpo/core/tor/-/commit/8ead53330c73e9bc1b82f6b7fc8946d629063842

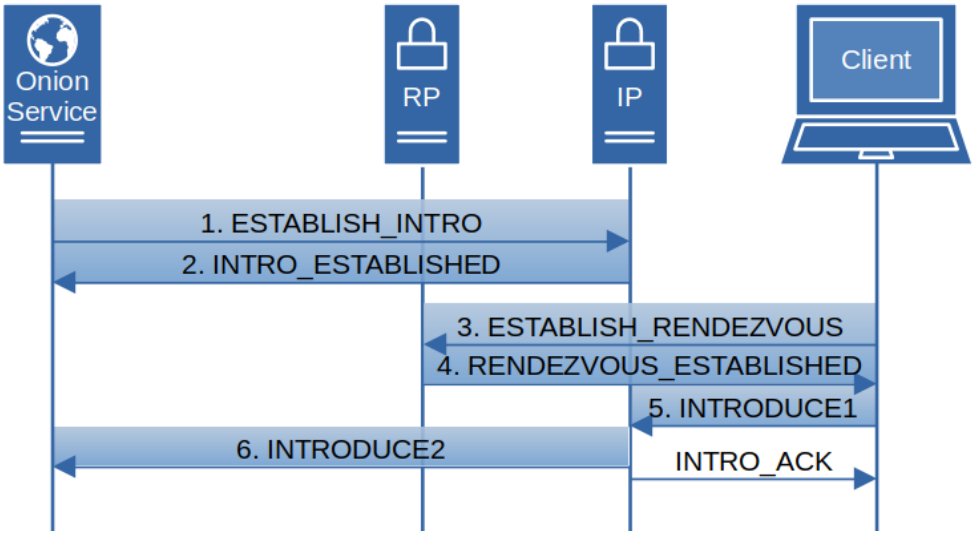

Figure 7.3: Circuit latency with our prototype

our own introduction point, because we were not interested in the individual timings for *INTRODUCE1* and *INTRODUCE2* requests.

For our measurements, we expect two Tor processes to be up and running. One of them acts as publisher and accepts incoming subscribe requests via an onion service, while the other acts as subscriber. Afterwards, a full Pub/Sub exchange is simulated:

1. The subscriber creates a new onion service.
2. The subscriber sends a subscribe message to the publisher.
3. The publisher receives the subscription.
4. The publisher publishes an event that the subscriber subscribed to.
5. The subscriber receives the event.

This enables us to quantify the relative performance gained by implementing the modifications from section 7.2.

Since it is our goal to quantify the performance improvement of our proposed changes individually, we decided that our prototype would only use the optimizations from section 7.3 and 7.4 when subscribing to events. This enables us to quantify the relative performance gained by implementing the modifications from section 7.2 by comparing the time needed to subscribe to an event with the time needed to publish one. All the data presented in the next section was collected over 10,000 runs between 2022-05-06 and 2022-05-11.

7.5.2 Experiment Results

Before presenting our results, it is worth discussing our expectations going into this experiment. The unmodified baseline should take a few seconds to create the onion service. Recent results [59] suggest waiting times of slightly below four seconds before the onion service is ready. The time it takes to subscribe to an event or publish a notification about an event should be equal, since both actions need to establish a fresh connection via onion services. When measured

Runs	Unmodified	Modified
Attempted runs	4948	5052
Successful runs	4822	4569
Bootstrap timeout	48	50
Descriptor parsing failed	0	132
Descriptor download failed	78	0
Run timed out	0	271
Control protocol timeout	0	31

Table 7.1: Number of conducted test runs

in 2008 [87], connecting to an onion service took on average 24 seconds. However, Tor has continued to improve since then with more efficient cryptography and protocols. That, in combination with the fact that the Internet as a whole has gotten faster, leads us to expect that the time needed to connect to an onion service should be much lower today.

The improvement presented in section 7.2 should significantly decrease the time it takes to create an onion service, while the time needed to subscribe to or publish an event should only decrease slightly. The modifications presented in sections 7.3 and 7.4 should massively decrease the time it takes to subscribe to an event, without impacting any other part of the communication. Figure 7.3 visualizes the expected improvement by showing the number of circuits needed to connect to an onion service after all proposed modifications have been applied.

Table 7.1 shows how many runs were conducted during the experiment and how many of them were successful. Only successful runs were included in further performance analysis, so the failed runs are broken down to clearly define the reasons why runs were excluded. First, runs where Tor needed more than 300 seconds for initial bootstrapping were aborted. This is independent of our modifications and occurs equally for both implementations. Our prototype introduces an expected issue with service descriptors not being parsed successfully, if the time period changes between the creation of a descriptor and the publication of the descriptor, which caused 132 additional failures. It is also not surprising that our prototype implementation does not possess the same resilience against common Tor issues—like circuits collapsing before they could be used—which resulted in a total of 271 failures. Not expected were the issues with failing descriptor downloads with regular onion services, which will be discussed in more detail below, and the control protocol timeouts that happened when creating an onion service took longer than 5 seconds.

A first overview over our baseline measurements is presented in Table 7.2, showing how long the individual stages of the Pub/Sub process took while using regular onion services. As expected, connecting to an onion service became much faster, but it still takes more than 8 seconds on average to exchange data with an onion service. Initially, it looked like the time needed to create an onion service is significantly below four seconds. Unfortunately, this result is not caused by an improvement in the Tor network, but by different interpretations of when an onion service has been published. Hoeller et al. [59] considered an onion service successfully published after the descriptor was uploaded to 8 out of 16 HSDIR nodes, while the Stem library used by us only requires a single upload to complete. This explains why the onion service creation time was lower than expected and why some of our experiment runs failed because a descriptor

Description	Minimum	Maximum	Median	Average	Variance	StdDev
Creating and publishing callback onion service	1.3322	26.2639	1.7956	2.0985	0.9469	0.9731
Sending subscribe to publisher	1.1526	208.9262	3.4638	8.4719	381.1696	19.5236
Sending event notification to subscriber	1.1253	206.3544	3.5483	8.5836	361.2280	19.0060
Total time	4.1885	272.0048	9.8988	19.4779	774.0817	27.8223

Table 7.2: Pub/Sub duration (in seconds) with unmodified baseline Tor implementation

Description	Minimum	Maximum	Median	Average	Variance	StdDev
Loading descriptor via control protocol	0.0019	0.0215	0.0041	0.0047	0.0000	0.0021
Creating callback onion service	1.0220	6.0388	1.0729	1.3544	0.3396	0.5827
Creating introduction circuit	0.0419	35.2306	0.3809	0.5949	2.0276	1.4239
Sending subscribe event to publisher	0.0465	29.7967	0.1770	0.2354	0.3171	0.5631
Sending event notification to subscriber	0.7203	58.3497	2.2798	3.1695	17.6772	4.2044
Total time	2.1152	60.8921	4.2544	5.3740	21.4259	4.6288

Table 7.3: Pub/Sub duration (in seconds) with modified Tor prototype

could not be downloaded. If the first descriptor is not used in the current time period, then the Tor client will fail to download the service descriptor, because it has not yet been published for the current time period. As shown in Table 7.1, within our 4948 runs we encountered this issue 78 times.

At this point we considered repeating our experiment to obtain more reliable results for onion service creation time. However, it turns out that it is actually quite hard to define when an onion service has been successfully published: Do all uploads have to complete? Or is it enough if half of them have completed? Should uploads for current and backup descriptor be weighted differently? Would that weighting depend on the current time, since backup descriptors are more likely to be used around time period changes? Depending on how one answers these questions, the time needed to publish an onion service can change drastically. Since we are trying to measure an objective improvement over the current Tor implementation, selecting the most favorable definition for Tor does seem fair and that happens to be what the Stem library is doing by requiring just a single upload to complete.

Putting that issue aside, it is worth highlighting that the median for connecting to an onion service is much lower than the mean, with only about 3.5 seconds. This indicates that the mean is increased by (potentially few) outlier requests needing a long time to finish, which is also confirmed by the high variance and standard deviation. This is most likely caused by circuits being built through unreliable or currently overloaded/unstable Tor relays and is inherent to the design of the Tor network. Nevertheless, such latency spikes are a problem in latency sensitive domains like instant messaging and discourage users from adopting privacy preserving applications build on top of the Tor network, so reducing them as much as possible should remain a goal.

We can now compare our improvements against this baseline of an unmodified Tor implementation. Table 7.3 shows the time needed by the new steps introduced by our prototype, as well as the old steps that should have been sped up. The overall trend is promising, with the median time needed for a full Pub/Sub exchange reduced from 9.9 to 4.3 seconds. A more detailed comparison of our measurements is presented in Figure 7.4.

The newly introduced function to load service descriptors via the control protocol barely has any impact on the overall timing, confirming that loading descriptors directly is feasible for applications that can distribute them. Removing the step of publishing service descriptors to the hidden service directory did improve the time needed to prepare an onion service. It should be noted though, that this time difference is most likely irrelevant in practice, because the time needed to distribute the service descriptor to potential clients is sure to be longer than the time needed for the descriptor to be published. Therefore, we consider the performance gained at this stage irrelevant for the overall performance of a tor-aware Pub/Sub implementation.

Of much greater relevance is the time needed for subscribing and publishing, as those are the operations that users are likely to be waiting for. Subscribing to an event was measured in two parts. First, we measured the time it takes to create a custom introduction circuit. This time is interesting, because our prototype implementation always creates a fresh introduction circuit, instead of trying to cannibalize an existing one first. So the time needed to create an introduction circuit also gives us an estimate on how long it takes Tor to create a new circuit from scratch. With an average duration of 0.6 seconds, the time needed to create the introduction circuit is more than double of what it takes to send the *INTRODUCE1* cell via the introduction point to the onion service, which requires going through two different circuits and only takes 0.24 seconds on average. This means that data bundled with the *INTRODUCE1* cell needs on average less

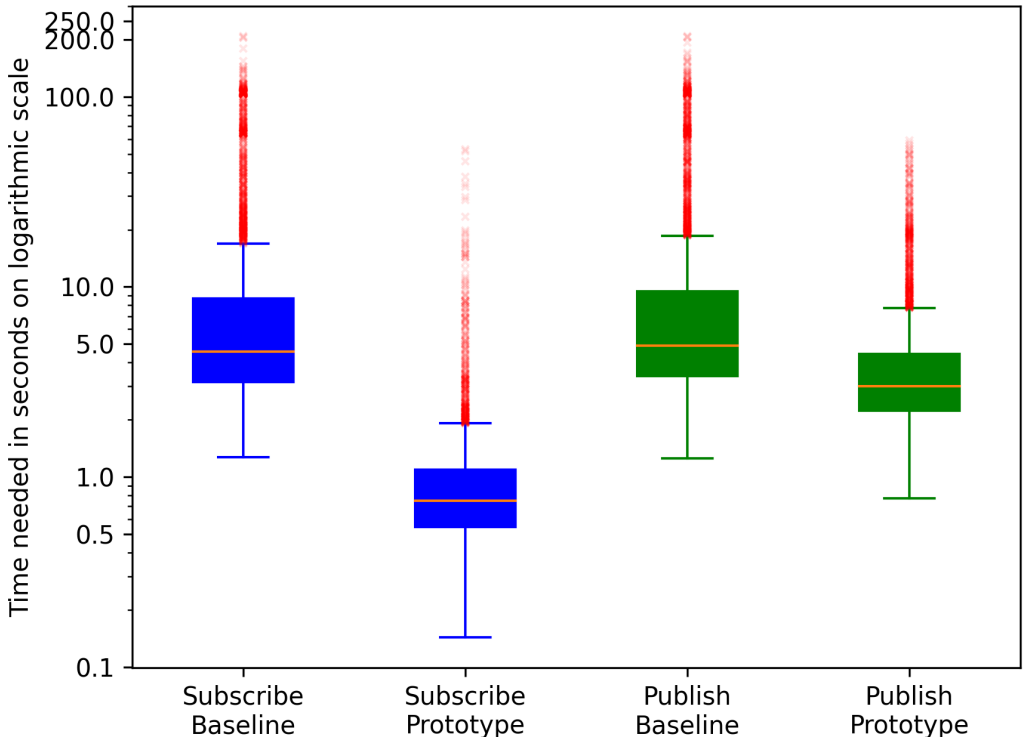

Figure 7.4: Comparison of Subscribe and Publish times

than a second to reach its destination, compared to more than 8 seconds if data is transmitted the traditional way. However, this comparison is not completely fair because this subscribe request also benefits from not having to communicate with the hidden service directory.

To evaluate the performance improvement by skipping the hidden service directory, the time needed for sending event notifications back to the subscriber can be analyzed. As expected, this request takes much longer than the subscription, because it establishes a regular rendezvous circuit. Our measurements show that the median time needed to send a subscription was reduced from about 3.5 seconds to only 2.3 seconds.

Especially interesting was the fact that the variance in our measured times was decreased significantly by our proposed modifications, although that was never an explicit goal. The variance for sending an event notification dropped from more than 361.2 seconds down to only 17.7 seconds. To rule out that the observed improvements were not caused by outliers within the significant variance of the baseline, we conducted an unequal variances t-test (Welch's test) for both the time needed to send an event notification to the subscriber and the total time to test the assumption that our prototype resulted in faster executions than the baseline. The results of these tests (provided in Table 7.4) do confirm that the observed time improvements are statistically significant, despite the high variance.

We originally speculated that this behavior is tied to our incorrect check for when onion services are published. If Tor has to connect to multiple nodes in the hidden service directory before finding the descriptor, the time needed to obtain the service descriptor becomes much less predictable. To confirm this theory, we repeated our experiment by conducting another 10,000 runs of our experiment between 2022-05-26 and 2022-06-06 where we added an arti-

Chosen test input	t-metric	p-value
Sending event notification	19.2889	$1.73 \cdot 10^{-80}$
Total time	34.6989	$2.43 \cdot 10^{-237}$

Table 7.4: Results of Welch's Tests on collected data

ficial delay of 10 seconds after the creation of the onion service. According to data collected in the past by Hoeller et al. [59] this should be enough time for most uploads to finish, but we did not observe a significant decrease in variance, indicating that our assumption was incorrect. This leads us to conclude that omitting the HSDIR has also decreased the variance in onion service connection times.

7.6 Summary

This chapter concludes the scientific contributions presented in this thesis by demonstrating several potential improvements that could be applied to the Tor network to significantly improve the performance of distributed systems running on top of the Tor network. Making the hidden service directory optional provides benefits both for performance and for privacy and can be done with minimal changes to the current Tor implementation. Using the initial introduction request to transfer additional information benefits applications that need to transfer lots of short messages to different recipients like the registration requests made by sensors. The final improvement makes these registration requests even more efficient by including a minimal service descriptor. This enables the sensor to send the responding match request without contacting the HSDir first, which again cuts down the response time by more than 50 % even if the messages are exchanged via a regular rendezvous circuit.

This resulted in a Tor version that can reduce the average time needed for the first half of a Digidow transaction from almost 20 seconds to slightly above 5 seconds. While this is still not enough to meet the objectives from section 1.2, it constitutes a significant improvement over the current state of the art and reveals paths towards further improvements in the future. Preparing onion services in advance or rotating them as suggested in section 5.3 would remove more than a second from the transaction time, and if match or credential requests could also be bundled within *Introduce1* cells, the transaction time would decrease even more drastically. Other use cases that require peer-to-peer connections via Tor—like privacy preserving instant messaging for example—also benefit from these optimizations because a lower number of circuits and cells needed leads to lower transmission times and less battery drain. The final chapter will provide a discussion on further research needed before a networking scheme for privacy preserving digital identity systems can be finalized.

Chapter 8

Conclusion and Outlook

8.1 Conclusion

After all the research presented in this thesis, it is time to check how close we are to meeting the objectives laid out in section 1.2. The first goal of having a distributed system can be achieved with the networking strategy presented in section 3. Although many details still require additional work, there is no doubt on our minds that the proposed networking strategy can achieve the desired attribute of distribution. The second requirement dealt with preserving the privacy of users. Our analysis in section 2.2 concluded that the Tor network is the only currently available system that can provide the network anonymity required to protect the privacy of users of a decentralized digital identity system. However, we have also identified several issues within the Tor network that reduce our trust. The HSDir leaking information about when and how often onion services are being used (see section 4), the missing transparency of directory authorities about their voting behavior (see section 6), and the fact that attackers have repeatedly managed to gain control over significant fractions of the Tor network [74, 99, 100] are all good reasons to remain careful when trusting entities like the Tor project. Nevertheless, the Tor project is still the best and most feasible effort towards anonymous networking, and we believe that building on top of it provides a very reasonable compromise between privacy, latency, and feasibility. The topic of scalability was not directly addressed very much during this thesis because fully decentralized systems have a tendency to scale very well. The biggest weakness in this regards is the Tor project itself because it is not fully decentralized. In 2009 McLachlan et al. [92] raised the issue that the Tor network is putting a lot of load on directory authorities because they have to distribute the Tor consensus to every client in the network. The Tor project has taken some measures to mitigate this issue—supporting consensus diffs [124] for example—and there are other ideas like the use of private information retrieval [93] to further reduce the scalability issues faced by the Tor network. As long as the Tor network manages to scale, we are confident that our proposed networking scheme will also scale. This brings us to the final objective: Keeping the latency of network transactions low enough to remain acceptable for users. We learned from section 7.5 that every interaction with an onion service takes more than 8 seconds. A minimal Digidow transaction (see section 3.1.5) consists of a registration, a match and a credential which would take more than 24 seconds in total. If the PIA manages to correctly register in advance, the time needed for the registration can even be dropped leaving only about 16 seconds for network latency. While this is below the maximum of 30 seconds for transaction times, there are many scenarios where such a long waiting time would be unacceptable. However, the improvements presented in section 7 show that it is possible to exchange the critical requests (the match and the credential) within less than two seconds instead. This improvement is enough to convince us that the network for a privacy preserving distributed digital identity system can be feasibly implemented today, if the privacy level

provided by the Tor network is acceptable for the specific scenario. There is a lot of implementation work that remains to be done, but all the necessary parts are available.

8.2 Future Work

Unfortunately, the research conducted during this thesis had a tendency to bring up more questions than it answered. Despite my best efforts, the time available for this dissertation is limited and many questions remain unanswered. This section will outline some research questions that will hopefully be addressed by other researches in the future to improve the tools and knowledge available to engineers who want to design privacy protecting technologies. Since the Tor project is currently undergoing an effort to re-implement the Tor application in the Rust programming language [90], it also seems like a good opportunity to propose changes that make the Tor network more useful for the future.

8.2.1 Improve I2P Metrics

The I2P network is—as discussed in section 2.3.1—designed to support anonymous distributed networks. However, its low number of users has not yet attracted enough outside review to justify trust into their network. One foundation to enable further research is to obtain better information about the current state of the I2P network. Efforts have been made to collect metrics [56] but there is a lot of information missing that is readily available for the Tor network. First, there is no information on client behavior or network performance. We have no statistics on how many connections are made via the I2P network, their available bandwidth, or how long it takes to establish them. Answering these questions is tricky because nodes within the I2P network are only aware of a small subset of other running nodes within the network. Obviously, the performance experienced by individual nodes is heavily impacted by the subset of the network they use to create their onion routed connections. To account for this, data must be collected on a large scale to derive meaningful averages and variances. Especially the variance would be interesting because it quantifies the randomness in performance introduced by the distribution of subsets instead of all nodes. There are also issues regarding the information available on the nodes that make up the network. Statistical information about them can be obtained by operating multiple I2P routers at once and combining their data. However, we only get an estimate on the amount of routers, their flags and their countries. Further relevant information like the AS (autonomous system) a node is located in, the version of I2P running on the router, their uptime, or something like a contact information are not available for I2P. This makes it harder for researchers to investigate detailed aspects of the I2P network because they have no data as a starting point to go from. Even worse, it also has a negative impact on the security of the network because the I2P network has no real way of preventing malicious nodes from joining. The much larger Tor network is regularly forced to remove relays from the network [74, 100], while I2P has no way of either detecting or removing large groups of malicious relays. As long as this remains the case, it is hard to justify trust in the I2P network.

8.2.2 Rework the Estimate for Unique V3 Onion Services

As discussed in section 4.2.1, the current estimates for unique V3 onion addresses published by the Tor project are calculated with the outdated HSDir shares that only apply for version 2. While this is one problem that can be addressed quite easily, there are a few other questions that arise from re-using the algorithm for the estimation of V2 onion services: First, the way how data is aggregated should be reviewed more carefully. V2 onion service counts were collected in 24 hours periods with random starting points, while V3 onion service counts use 24 hour periods with a fixed starting point. This means that the current algorithm goes through a lot of effort to account for overlaps between the estimates of different relays, although the estimates for V3 should not overlap anymore. The second question deals with the aggregation of *hsdir_shares*. This is necessary because a new consensus (and with it a new HSDir) is published every hour but the data reported by relays is aggregated by days. In theory, this would be a simple average over the 24 fractions calculated for the 24 different consensuses within the time period, but this is again not enforced because support for random starting times means that periods will usually have 25 valid consensuses. This issue is especially problematic because some relays report statistics that do not fit the specification. The collection period of reported statistics is often larger than the expected 86,400 seconds. Slightly higher values are negligible (the calculation already calculated with 25 consensuses instead of 24), but some relays report ridiculous data collection periods of more than 4000 days instead of just 24 hours. Other relays report wildly different estimates for the same time period, making it hard to decide which estimate is more legitimate. The current algorithm includes all of these reports into its aggregation. It is uncertain how much of an impact they have on the overall estimation, but investigating why these reports are made and how they should be handled to produce accurate statistics would definitely be worth additional research time. Finally, the chosen method of aggregating the reports from different relays into a single estimate should also be discussed. This currently happens with a weighted interquartile mean that discards estimates from relays whose share is either in the first or fourth quartile. A weighted mean is then calculated from the remaining estimates, with each estimate being weighted by its share. This prevents relays with very small or very high shares from impacting the overall result too much. This was most likely included because the V2 specification for onion services used the fingerprint of a relay to position it on the HSDir and a relay can choose its fingerprint freely, enabling the operators of Tor relays to control their share of the HSDir. Ignoring values reported by both very small and very large shares makes it harder for single entities to falsify the results. However, V3 onion services are no longer suffering from that issue because the share a relay is responsible for changes daily and is no longer predictable. Filtering statistics based on their total impact on the onion service instead of their weight might lead to more accurate predictions for the amount of available v3 onion services.

8.2.3 Compare Downloads of Known and Unknown Onion Services

The analysis presented in section 4.2.2 is very superficial and does not even attempt to a find out what onion services are being used for. The introduction of key blinding makes it impossible to reproduce previous research on V2 onion services—like the work of Owen and Savage [102]—because the onion addresses can no longer be harvested in this manner. If provided with a list of valid onion addresses, it would be easy to retroactively calculate their blinded public keys and find out how often they were observed by the experiment from

section 4.1 and make an estimated guess about the amount of visitors they receive. There are public resources that collect onion service addresses [62, 97] that could be used to create a list of known onion addresses. These are very likely only a tiny fraction of all existing onion addresses but they should be responsible for a significant share of onion service downloads. It would be interesting to see how much onion service activity can be attributed to these known onion services and how much remains unaccounted for. This could also serve as a baseline for further research into how onion addresses are distributed. After all, onion services delegate the distribution of onion addresses to their operators, so successful onion services must have a way for users to learn their contact information. If onion services with thousands of users are not included in public resources, they must either have currently unknown ways of onion address distribution or utilize onion services in a fashion where a few individuals cause a large amount of descriptor downloads. In any case, it seems likely that further analysis of the data harvested during our experiment will provide more and new insights into the usage of V3 onion services.

8.2.4 SingleHopOnionService vs. Public Service via Tor

A discussion that was shortly addressed in section 3.6.2 is the trade-off between running a service with a public IP address and expecting clients to connect via the Tor network and operating an onion service in *HiddenServiceSingleHopMode*. This mode was designed for service providers that wish to operate an onion service but do not need to keep their identity private [49]. It improves the performance of such onion services by shortening the length of circuits to a single hop instead of three. While it is obvious that this will improve the performance of an onion service, there is little research available on how significant that benefit is and more specifically, if it outperforms public services being accessed via onion services. There is some public data available on how the service latency is affected [36] but this was specifically measured for VOIP and cannot be easily generalized. A single-hop onion service connection consists of four hops, three from the client and one from the server but does not include an *exit* node. Considering that *exit* nodes are the most problematic to operate, it seems like a reasonable assumption that they are more likely to become a bottleneck during data transfer, so removing them from the equation could be beneficial. However, that benefit is immediately offset by the fact that a single-hop onion service connection still requires four hops instead of three, which increases the chance for one of the relays to be overloaded as well as overall latency by adding another indirection step. Another point to consider is that *exit* nodes might also add random latency on their path to the destination server, which is highly dependent on the location of the exit node.

Aside from the latency during an established connection, the time needed to establish a connection would also be an interesting point of comparison. Tor onion services require connections to the HSDir and rendezvous point while *exit* nodes need to do DNS resolution before they can continue to establish their connections. It would be interesting to measure how much of the onion service overhead is offset by *exit* nodes being overloaded. Finally, these measurements could also include the optimizations presented in chapter 7 to quantify the potential performance gains that could be achieved for distributed systems that only need to preserve the anonymity of a part of their members.

8.2.5 Evaluate Service Descriptor Lifetime

Another interesting question that arises when exchanging service descriptors directly (as proposed in section 7.2) is the one about the expected lifetime of service descriptors. While the encrypted service descriptor changes every 12 hours, the decrypted information about introduction points can remain valid for much longer. As long as the onion service is still connected to one of the introduction points specified within a descriptor, that descriptor can still be used to connect to the onion service, even if it has already expired. This leads to the question of how frequently a distributed system that exchanges service descriptors directly (like the sensor directory) would have to update their data to remain functional. Tor has selected a period of between 60 and 120 minutes for this purpose, but there is no research on the lifetime of service descriptors to confirm that this is indeed necessary. Based on the data presented in section 4.2.1 it should be possible to predict how often service descriptors are changed in practice. However, changing a single introduction point does not mean that clients with an outdated descriptor cannot connect anymore. If the Tor client fails to connect to one of the introduction points listed within a service descriptor, it continues to try the other introduction points and only fails if none of them are operational. So, as long as one of the introduction points in the current descriptor is also present in an outdated service descriptor, the outdated service descriptor can still be used to connect to the service. Measuring the average lifetime after which a service descriptor cannot be used to connect to an onion service any more would be quite interesting for the design of descriptor distribution schemes. The question becomes even more complicated when minor Tor changes are considered. A service descriptor can contain up to 20 different introduction points, which would massively increase the probability of at least one of those introduction points remaining available for extended periods of time. Alternatively, onion services could limit the candidate pool for introduction points to only stable relays. At least in theory, this should result in service descriptors with significantly increased lifetimes. Finally, the policy of creating a new introduction point as soon as a currently running one fails could be modified to give a relay that is only offline for a short period of time (for a restart for example) the opportunity to become available again. By creating a new introduction circuit to the same relay with the same *INTRO_AUTH_KEY*, the onion service could recover from the restart of a Tor client without a need to update/change their service descriptor.

8.2.6 Harden Onion Services against DDOS Attacks

A very active field of current research is the challenge of hardening onion services against distributed denial-of-service attacks [5]. The Tor project has already taken steps to mitigate the issue by implementing an extension [50, 127] that enables introduction points to drop connections early. Additionally, tools like onionbalance [16] can be used to distribute onion services and increase the volume of traffic they can handle. Unfortunately, both of these defenses have their issues: Telling *IP*s to drop requests after a certain threshold prevents the onion service from being overwhelmed but it also prevents valid users from connecting to the service, which was most likely the goal of the attacker. Increasing the traffic the onion service can handle makes attacks harder, but adversaries with enough computation power can still overwhelm the onion service. The key reason for this weakness lies in the design of onion services. Every valid *INTRODUCE2* cell triggers the creation of a new rendezvous circuit to a destination controlled by the attacker. If the attacker is only interested in overwhelming the onion service, it won't even connect to the rendezvous point it-

self, just forcing the victim to create a new circuit is enough. This is especially critical because the amount of outgoing circuits that can be created within the current Tor network is limited because the current Tor implementation never uses the same relay for two different circuits in parallel. While the attacker can create one circuit after the other, the onion service has to make sure that the *INTRODUCE2* cell is invalid every single time. This unbalanced workload is the core reason why denial-of-service attacks against onion services have remained feasible.

The Tor project currently investigates two approaches to resolve this situation [5]. Either by the use of digital credentials like PrivacyPass [25] that issue anonymous tokens to Tor clients for good behavior and allow them to use this tokens afterwards to access services under attack or by introducing a new extension that allows onion services to ask clients for a proof-of-work before it creates the rendezvous circuit [69]. Both of these ideas bring their own set of challenges, like what behavior should award tokens or how hard should the proof-of-work be, which have so far prevented the Tor project from implementing them. More research into this area is necessary to identify the best approach to tackle this issue. Within the network architecture proposed in chapter 3 the sensor is the device most vulnerable to such attacks. Its contact information is publicly available and successful attacks would have significant impact on both users and businesses, enabling attackers to blackmail sensor operators for not overloading their sensors.

8.2.7 Unlinkable Service Descriptors

As discussed in section 3.6.5, unlinkability between multiple registration requests made by a PIA to different sensors can be achieved by creating separate onion services. Technically, it is not necessary to create new onion services, it would be sufficient to create new introduction points, as they are the ones that provide linkability via their *INTRO_AUTH_KEYs*. With the optimization presented in section 7.2, establishing new introduction points is by far the most time consuming part of creating new onion services, so this distinction is not particularly helpful on its own. However, it leads to the conclusion that changing the *INTRO_AUTH_KEY* to no longer uniquely identify an onion service would enable onion services to hand out multiple service descriptors that all lead to the same destination. To be fair, the *ENC KEY* provides just as much linkability as the *INTRO AUTH KEY* but an onion service can simply create multiple *ENC KEYs* and attempt decryption with all of them, so they are not as much of a challenge as the *INTRO AUTH KEYs* are. Making them unlinkable requires a change to the logic for introduction points that would have to propagate throughout the Tor network before it could be widely adopted. This could for example be achieved by the same key blinding mechanism that is used to create blinded public keys to identify service descriptors. Instead of including the *INTRO AUTH KEY* directly in the service descriptor, a service descriptor could contains a *blinded intro auth key* along with the NONCE used to create that key. Attackers with access to multiple service descriptors would no longer be able to realize if two service descriptors were related or not, but the introduction point could still identify the correct introduction circuit if provided with both the NONCE and the *blinded intro auth key* by simply blinding all *INTRO AUTH KEYS* it currently knows with the NONCE and comparing them to the requested *blinded intro auth key*.

The interesting research question to answer is if this is sufficient to render service descriptors unlinkable. After all, attackers could still identify the Tor relay that is used as an introduction point, they would just be unable to confirm that

the introduction point is forwarding all of those requests to the same onion service. Onion services with multiple introduction points are likely to remain identifiable just based on the identities of their introduction points. Service descriptors that only contain a single introduction point would also suffer from this problem, but they would be much harder to correlate. While there are no public estimates on the number of onion services that an average relay acts as introduction point for, we do know that there are less than 7000 Tor relays in total that service more than 600.000 onion services. Calculating with an average of three introduction points per onion service, the average Tor relay has more than 250 active introduction circuits open. In practice, it seems reasonable to assume that long-lived relays are handling many more introduction point connections than relays that are restarted regularly, but there is no public data available on that topic. This might be enough noise to prevent attackers from easily finding out if service descriptors belong to the same onion service by just looking at the identity of the introduction point. Statistical evaluations on how much easier this gets when service descriptors include more than one introduction point would be necessary to decide if this approach might be useful for future Tor versions. At the same time it should be evaluated if blinding every *INTRO AUTH KEY* for every introduction request is too time consuming to be applied in practice.

8.2.8 Optimize Space in Introduce1 Cells

A core limitation of the improvement presented in section 7.3 is the space available for additional messages. As already hinted at in section 7.3.2, there are several ideas how the available space within an *INTRODUCE1* cell could be used more efficiently. In order to remain compatible with the current Tor network, an *INTRODUCE1* cell always has to have a size of 512 bytes, so this constitutes a theoretical upper limit on the data available for transmission. Since the *INTRODUCE1* cell allocates 14 bytes for headers, 498 bytes remain available to transmit data. The necessary data can logically be divided into two categories (cmp. Figure 2.5): Unencrypted information for the introduction point and encrypted information for the onion service itself. The introduction point currently expects 55 bytes of data in order to forward the cell to the onion service. However, 20 of those 55 bytes are still reserved for a legacy key that is no longer needed, because relays old enough to rely on it are no longer allowed to be part of the Tor network [48, 52, 73]. Unfortunately, dropping the legacy key would be a breaking change because current Tor versions expect this key to be included in the *INTRODUCE1* cell, which is why we did not include this in our prototype implementation. It should however be removed in the future as it does not serve any purpose currently. This leaves a maximum of 463 bytes available for the content of the *INTRODUCE2* cell (once the legacy key has been removed). For the current version, only 443 bytes remain available. The optimal utilization of the data forwarded within the *INTRODUCE2* cell could be achieved by defining a new cell format that drops all information about the rendezvous point from the cell. The less optimal yet more generic alternative is to utilize regular introduction cells and Tor extensions—as our prototype for chapter 7.3 does—to forward information directly from a client to the onion service. In this case, space is used less efficiently because onion services expect to find all the information needed to connect to a rendezvous point, even if the extension tells them not to connect to it. As demonstrated in section 7.4, there is room for improvement in this area as well, by encoding other information (like parts of a minimal service descriptor) in fields that usually provide information about the rendezvous point. Unused fields, like the rendezvous cookie and onion key, could be used to transfer an additional 52 bytes of data to the onion service.

Finally, it is worth discussing how Tor uses link-specifiers to identify nodes within the network. They are included in service descriptors to disclose the identity of introduction points or in *INTRODUCE2* requests to tell the onion service which node acts as rendezvous point. There are four different ways how relays can be identified:

1. IPv4 address + port (4+2 bytes): Must be included in every onion service.
2. RSA fingerprint (20 bytes): Fingerprint of the relays RSA identity key. Must always be included.
3. Ed25519 identity key (32 bytes): Complete identity key of the relay. Must always be included because there are no V3 onion service implementations that do not support ed25519 keys.
4. IPv6 address + port(16+2 bytes): Should be included, if applicable, but is not mandatory.

Since every single link specifier needs to specify a type and length to be parsed correctly, the total size of a complete link specifier should sum up to $8 + 6 + 20 + 32 + 18 = 84$ bytes. However, the actual payload is larger because a Tor struct always allocates enough space to handle any potential input and the largest potential input would be the ed25519 key. This causes $4 * (2 + 32) = 136$ bytes of memory to be allocated for the link specifiers and because the specification requires link-specifiers to be transmitted in base64 encoding, the total size they take up within the *INTRODUCE2* cell actually ends up around 180 bytes. It is understandable that Tor used redundant link specifiers to ensure that every relay on a circuit understands at least one of them, but considering that every supported Tor version understands ED25519 identity keys [52] and unsupported versions are not allowed to join the Tor network anymore [48, 73], the inclusion of this additional identity information seems no longer useful today. Allowing introduction requests to only include the ED25519 key as link-specifier would save about 130 additional bytes of data that could be made available for transactions, which definitely seems like a worthwhile improvement. The Tor project might even want to consider dropping the need for alternative link-specifiers entirely in future versions.

8.2.9 Encode Minimal Service Descriptors in Hostnames

The minimized service descriptors presented in section 7.4 are actually short enough to be encoded with less than 256 bytes, which is the maximum hostname length supported by SOCKS [81]. This would in theory allow the creation of hostnames that encode all the information needed to connect to them within their own name. Tools like OnionShare [80] that cannot reasonably forward service descriptors because they make no assumptions on how the share link is forwarded, might find this a viable approach to circumvent the HSDir and reap the performance and privacy benefits that can be obtained by doing so. The only downside is that onion addresses become significantly longer, but onion addresses are already too long to be typed conveniently, resulting in copy & paste being the preferred mode of exchanging them. When copying a URL or clicking on a link, the length of the hostname is irrelevant and even in situations where it is not, the use of url shortener services can be used to work around the issue (at the cost of leaking information to a third party). Implementing this functionality would require the logic within the Tor SOCKS proxy that interprets and parses regular onion addresses to also support service descriptors directly encoded within the hostname. The engineering effort required to achieve this goal would be fairly limited if the Tor project ever decides to adopt the concept of minimized service descriptors.

8.2.10 Minor Implementation Improvements

Distribute unencrypted descriptors

In order to reliably bypass the hidden service directory, it should be possible to load descriptors in an unencrypted form to avoid timing issues during time period changes. It might make sense to also publish them in an unencrypted format, which would require the definition of a new control protocol event. This is not strictly necessary, since decryption is straight forward when the time period is well defined.

Make PublishHidServDescriptors a per-service option

Right now, a single Tor instance can either publish all its onion services or none of them, but some scenarios might benefit from being able to selectively publish individual onion services. This could be achieved by setting the *PublishHidServDescriptors* option per onion service rather than globally. We did not address this issue in our work, because the control protocol does support manually publishing service descriptors, so for prototypes this use case can already be implemented.

8.3 Epilogue

This thesis documents a 4-year long research endeavor that started with a simple questions (How to achieve network privacy in a distributed digital identity system) and ended with multiple extensions to Tor onion services. We started with a detailed investigation in the technologies already available in section 2.2 before deciding how to continue. For chapter 4, we designed a real world experiment that required great care to not endanger users, interacted with an ethics board to double check our design, and finally ran the experiment for several months to learn more about the Tor network. We proved our capability to independently analyze large amounts of data by extracting numerous interesting statistics about the current usage of V3 onion services from our collected data. Notably, we presented the first public estimate on the number of V3 onion services operated within the Tor network. This estimate later became even more interesting when Tor metrics published their own estimates which did not match with our estimates at all. As a researcher, I do not consider this to be an issue at all. The scientific process asks researchers to obtain knowledge in a structured process that can be documented and reproduced by others to enable reproduction and verification of knowledge. While the original estimate on the number of V3 onion services was wrong, the wrong formula was publicly documented and published at FOCI'21 [60] after passing peer-review (showing that even peer-review cannot guarantee correct results). Thankfully, the Tor Metrics team had documented their calculation publicly, enabling us to identify our own errors and find some errors within their calculation, resulting in a more accurate estimate on the total number of onion services than before. This stands as an excellent example for the scientific process working as intended.

Chapter 5 saw us conducting yet another experiment on the live Tor network. Contrary to the experiment in chapter 4 where we had no specific question in mind, this experiment was specifically designed to answer one question: How

long does it take to deploy a V3 onion service? Our experiment successfully provided the information necessary to answer that question and provided insight into which parts of creating an onion service are the most time consuming. This information directly resulted in the decision to search for ways to circumvent the hidden service directory in certain situations.

Probably most interesting from a research perspective, is the work documented in chapter 6. It was never expected that this issue would be investigated during research into using onion services for distributed systems. The maintenance of multiple Tor relays for the experiment in chapter 4 raised multiple questions into how the Tor consensus was being formed. We tried to understand these questions and ended up confirming that the Tor network is not always acting according to its own public specification. During this investigation, we also noticed a group of relays displaying a pattern that piqued our curiosity once again. Their decision to restart at the beginning of every month seemed harmless enough, there were several reasonable explanations for such a behavior, and it did not seem relevant for the research question addressed in this theses, so there was no real need to follow up on this question. Our persistence was rewarded, as we were able to confirm that all of these relays were actually operated by a single (most likely malicious) entity and they were removed from the Tor network once the Tor project had successfully reproduced our results. Admittedly, this did not contribute to answering our research questions, but it made every single user of the Tor network safer than they had been before. Beyond that, it proves that another important aspect of the scientific process was observed during the research for this thesis. Experiments should always be conducted with specific questions, but not specific answers in mind. In our case, conducting an experiment on onion services led us to analyze the Tor consensus and identify a group of malicious relays. It should also be noted, that the Tor project once again showed their own commitment to the scientific process by reproducing our results—which was easy to do because we provided them with a structured way of doing so—and only taking action after they had reached the same conclusions as us.

Chapter 7 stands somewhat apart as the chapter where we stopped learning about the state of the current Tor network and progressed to proposing changes based on the results of our previous research. The research in chapter 5 had told us that onion services spent most of their creation time on creating circuits and most circuits were only created to upload to the hidden service directory. This directly inspired the idea to bypass the HSDir (cf. section 7.2) and reduce the number of circuits needed by directly forwarding information inside the *INTRODUCE1* request (cf. section 7.3). It seemed like a reasonable theory that combining those two improvements (cf. section 7.4) would result in significant latency gains that should improve user experience significantly. Once again, this assumption was confirmed by an experiment conducted on the live Tor network to quantify the performance gained by our proposed changes. The results were more than satisfying and we do expect the Digidow project to try and continue building on this foundation in the future.

Unfortunately, the time available for a dissertation is limited and research often has a habit of raising more new questions than it answers. This also held true for this research project and section 8.2 on future work is full of unanswered questions and further ideas that could and should be worked on in the future. The goal of this thesis is to demonstrate that the author is capable of conducting scientific research on his own and the presented work is hopefully sufficient to come to that conclusion.

Bibliography

[1] Shweta Agrawal, Subhashis Banerjee, and Subodh Sharma. 2017. Privacy and Security of Aadhaar A Computer Science Perspective, (September 2017).

[2] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. 2017. MCMix: Anonymous Messaging via Secure Multiparty Computation. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, (August 2017), pp. 1217–1234. ISBN: 978-1-931971-40-9. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/alexopoulos.

[3] Smart Card Alliance. 2011. Transit and Contactless Open Payments: An Emerging Approach for Fare Collection. Whitepaper TC-11002. Smart Card Alliance, (November 2011). https://www.securetechalliance.org/resources/pdf/Open_Payments_WP_110811.pdf.

[4] Pieter Arntz. 2021. Was threat actor KAX17 de-anonymizing the Tor network? (December 2021). Retrieved 09/26/2022 from https://www.malwarebytes.com/blog/news/2021/12/was-threat-actor-kax17-de-anonymizing-the-tor-network.

[5] asn. 2020. Retrieved 08/30/2022 from https://blog.torproject.org/stop-the-onion-denial/.

[6] asn. 2018. Announcing the Vanguards Add-On for Onion Services. Retrieved 08/06/2022 from https://github.com/mikeperry-tor/vanguards.

[7] Felipe Astolfi, Jelger Kroese, and Jeroen Van Oorschot. 2015. I2P - The Invisible Internet Project. Web Technology Report. Media Technology, Leiden University. https://staas.home.xs4all.nl/t/swtr/documents/wt2015_i2p.pdf.

[8] James Ball, Bruce Schneier, and Glenn Greenwald. 2013. NSA and GCHQ target Tor network that protects anonymity of web users. (October 2013). Retrieved 07/28/2022 from https://www.theguardian.com/world/2013/oct/04/nsa-gchq-attack-tor-network-encryption.

[9] Lamiaa Basyoni, Noora Fetais, Aiman Erbad, Amr Mohamed, and Mohsen Guizani. 2020. Traffic Analysis Attacks on Tor: A Survey. In *2020 IEEE International Conference on Informatics, IoT Enabling Technologies (ICIoT)*, pp. 183–188. DOI: 10.1109/ICIoT48696.2020.9089497.

[10] Patrick Beuth. 2021. Im Tor-Netzwerk hat sich ein unbekannter Beobachter ausgebreitet. (December 2021). Retrieved 09/26/2022 from https://www.spiegel.de/netzwelt/web/mysterium-kax17-im-tor-netzwerk-hat-sich-ein-unbekannter-beobachter-ausgebreitet-a-9891a67e-303d-4252-a1af-87d9c87407ec.

[11] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. 2013. Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (*SP '13*). IEEE Computer Society, USA, pp. 80–94. ISBN: 9780769549774. DOI: 10.1109/SP.2013.15. https://doi.org/10.1109/SP.2013.15.

[12] Blueprint for Free Speech. [n. d.] Ricochet Refresh. Retrieved 09/24/2022 from https://www.ricochetrefresh.net.

[13] Blueprint for Speech. 2019. Recichet Refresh. Retrieved 08/06/2022 from https://www.ricochetrefresh.net.

[14] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. 2007. Denial of Service or Denial of Security? How Attacks on Reliability can Compromise Anonymity. In *Proceedings of CCS 2007*. Association for Computing Machinery, (October 2007), pp. 92–102. ISBN: 9781595937032. DOI: 10.1145/1315245.1315258.

[15] John Brooks. [n. d.] Ricochet. Retrieved 09/24/2022 from https://ricochet.im.

[16] Donncha Ó Cearbhaill and George Kadianakis. 2015. onionbalance. Retrieved 09/06/2022 from https://github.com/asn-d6/onionbalance.

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

[18] Chen Chen, Daniele Asoni, David Barrera, George Danezis, and Adrain Perrig. 2015. HORNET: High-Speed Onion Routing at the Network Layer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (*CCS '15*). Association for Computing Machinery, Denver, Colorado, USA, pp. 1441–1454. ISBN: 9781450338325. DOI: 10.1145/2810103.2813628. https://doi.org/10.1145/2810103.2813628.

[19] Muqian Chen, Xuebin Wang, Tingwen Liu, Jinqiao Shi, Zelin Yin, and Binxing Fang. 2019. SignalCookie: Discovering Guard Relays of Hidden Services in Parallel. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1–7. DOI: 10.1109/ISCC47284.2019.8969639.

[20] Muqian Chen, Xuebin Wang, Jinqiao Shi, Can Zhao, Meiqi Wang, and Binxing Fang. 2020. Napping Guard: Deanonymizing Tor Hidden Service in a Stealthy Way. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 699–706. DOI: 10.1109/TrustCom50675.2020.00097.

[21] Catalin Cimpanu. 2021. A mysterious threat actor is running hundreds of malicious Tor relays. (December 2021). Retrieved 09/26/2022 from https://therecord.media/a-mysterious-threat-actor-is-running-hundreds-of-malicious-tor-relays/.

[22] Alissa Cooper, Hannes Tschofenig, Bernard Aboba, Jon Peterson, John Morris, Marit Hansen, and Rhys Smith. 2013. Privacy Considerations for Internet Protocols. RFC 6973. RFC Editor, (July 2013). https://datatracker.ietf.org/doc/html/rfc6973.

[23] David Cooper, Stefan Santesson, Sharon Boeyen, Russel Housley, and Tim Polk. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC. RFC Editor, (May 2008). https://www.ietf.org/rfc/rfc5280.txt.

[24] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. 2015. Riposte: An Anonymous Messaging System Handling Millions of Users. In *2015 IEEE Symposium on Security and Privacy*, pp. 321–338. DOI: 10.1109/SP.2015.27.

[25] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. 2018. Privacy Pass: Bypassing Internet Challenges Anonymously. *Proc. Priv. Enhancing Technol.*, 2018, 3, 164–180.

[26] Claudia Diaz, Harry Halpin, and Aggelos Kiayias. 2021. The Nym Network. Whitepaper 1. Nym Technologies SA, (February 2021). https://nymtech.net/nym-whitepaper.pdf.

[27] Roger Dingledine. 2015. Did the FBI Pay a University to Attack Tor Users? (November 2015). Retrieved 07/29/2022 from https://blog.torproject.org/did-fbi-pay-university-attack-tor-users/.

[28] Roger Dingledine. 2021. Exit relay operators please help test #2667 branch. (January 2021). Retrieved 08/07/2022 from https://lists.torproject.org/pipermail/tor-relays/2021-January/019258.html.

[29] Roger Dingledine. 2013. Improving Tor's anonymity by changing guard parameters. (October 2013). Retrieved 08/01/2022 from https://blog.torproject.org/improving-tors-anonymity-changing-guard-parameters/.

[30] Roger Dingledine. 2014. Tor security advisory: "relay early" traffic confirmation attack. (July 2014). Retrieved 07/29/2022 from https://blog.torproject.org/tor-security-advisory-relay-early-traffic-confirmation-attack/.

[31] Roger Dingledine. 2022. We're trying out guard-n-primary-guards-to-use=2. (July 2022). Retrieved 08/01/2022 from https://forum.torproject.net/t/tor-relays-were-trying-out-guard-n-primary-guards-to-use-2/3790.

[32] Roger Dingledine, Nicholas Hopper, George Kadianakis, and Nick Mathewson. 2014. One fast guard for life (or 9 months). In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2014)*. (July 2014). https://petsymposium.org/2014/papers/Dingledine.pdf.

[33] Roger Dingledine and Nick Mathewson. 2003. Tor Protocol Specification. Retrieved 07/29/2022 from https://github.com/torproject/torspec/blob/main/tor-spec.txt.

[34] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *13th USENIX Security Symposium (USENIX Security '04)*. https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router. USENIX Association, San Diego, CA, (August 2004).

[35] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. 2004. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *International conference on the theory and applications of cryptographic techniques*. Springer, pp. 523–540.

[36] Quentin Dufour. 2021. *High-throughput real-time onion networks to protect everyone's privacy*. PhD thesis. http://www.theses.fr/2021REN1S024/document. 2021REN1S024.

[37] Roya Ensafi, Philipp Winter, Abdullah Mueen, and Jedidiah R Crandall. 2015. Analyzing the Great Firewall of China Over Space and Time. *Proc. Priv. Enhancing Technol.*, 2015, 1, 61–76.

[38] European Comission. 2022. Proposal for a REGULATION OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL laying down rules to prevent and combat child sexual abuse. (May 2022). Retrieved 07/25/2022 from https://ec.europa.eu/home-affairs/proposal-regulation-laying-down-rules-prevent-and-combat-child-sexual-abuse_en.

[39] Directorate-General for Migration European Commission and Home Affairs. 2014. *Technical study on smart borders : final report*. Publications Office, (October 2014). DOI: doi/10.2837/86143.

[40] European Parliament and Council. 2014. REGULATION (EU) No 910/2014 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC. (June 2014). Retrieved 07/22/2022 from http://data.europa.eu/eli/reg/2014/910.

[41] Toralf Förster. 2021. Questions about consensus votes. (April 2021). Retrieved 08/07/2022 from https://lists.torproject.org/pipermail/tor-relays/2021-April/019604.html.

[42] Mengle Gautam. 2019. Major Aadhaar data leak plugged: French security researcher. Retrieved 07/26/2022 from https://www.thehindu.com/sci-tech/technology/major-aadhaar-data-leak-plugged-french-security-researcher/article26584981.ece.

[43] David Goldschlag, Michael Reed, and Paul Syverson. 1999. Onion routing. *Communications of the ACM*, 42, 2, (February 1999), 39–41. DOI: doi/10.1145/293411.293443. https://dl.acm.org/doi/pdf/10.1145/293411.293443.

[44] Google. 2014. FCM Architectural Overview. Retrieved 10/06/2022 from https://firebase.google.com/docs/cloud-messaging/fcm-architecture.

[45] Google and Apple. 2020. Exposure Notification – Bluetooth® Specification. whitepaper Version 1.2. (April 2020). https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ExposureNotification-BluetoothSpecificationv1.2.pdf.

[46] David Goulet. 2018. Bug26367 035 01. (July 2018). Retrieved 08/30/2022 from https://github.com/torproject/tor/pull/218.

[47] David Goulet. 2020. Onion Service version 2 deprecation timeline. Retrieved 07/29/2022 from https://blog.torproject.org/v2-deprecation-timeline/.

[48] David Goulet. 2019. Removing End-Of-Life Relays from the Network. (October 2019). Retrieved 08/08/2022 from https://blog.torproject.org/removing-end-life-relays-network/.

[49] David Goulet. 2016. What's new in Tor 0.2.9.8? (December 2016). Retrieved 08/12/2022 from https://blog.torproject.org/whats-new-tor-0298/.

[50] David Goulet and Roger Dingledine. 2019. ESTABLISH_INTRO Cell DoS Defense Extension. Retrieved 09/06/2022 from https://github.com/torproject/torspec/blob/main/proposals/305-establish-intro-dos-defense-extention.txt.

[51] David Goulet, Aaron Johnson, George Kadianakis, and Karsten Loesing. 2015. Hidden-service statistics reported by relays. Technical report 2015-04-001. The Tor Project, (April 2015). https://research.torproject.org/techreports/hidden-service-stats-2015-04-28.pdf.

[52] David Goulet, Nick Mathewson, and George Kadianakis. 2022. CoreTorReleases. (April 2022). Retrieved 08/08/2022 from https://gitlab.torproject.org/tpo/core/team/-/wikis/NetworkTeam/CoreTorReleases.

[53] Government of India. 2009. Unique Identification Authority of India. (January 2009). Retrieved 07/25/2022 from https://uidai.gov.in/.

[54] Sebastian Hahn. 2021. Questions about consensus votes. (April 2021). Retrieved 08/07/2022 from https://lists.torproject.org/pipermail/tor-relays/2021-April/019603.html.

[55] Harry Halpin and Ania Piotrowska. 2022. Achieving Network Privacy In Bitcoin: VPNs And Tor Help, But Mixnets Are Needed. (January 2022). Retrieved 07/27/2022 from https://bitcoinmagazine.com/technical/why-mixnets-are-needed-to-make-bitcoin-private.

[56] Nguyen Phong Hoang, Panagiotis Kintis, Manos Antonakakis, and Michalis Polychronakis. 2018. An Empirical Study of the I2P Anonymity Network and Its Censorship Resistance. In *Proceedings of the Internet Measurement Conference 2018* (*IMC '18*). ACM, Boston, MA, USA, pp. 379–392. ISBN: 978-1-4503-5619-0. DOI: 10.1145/3278532.3278565. http://doi.acm.org/10.1145/3278532.3278565.

[57] Tobias Höller. 2022. Estimate for V3 onion services uses network fractions for V2. (August 2022). Retrieved 09/02/2022 from https://gitlab.torproject.org/tpo/network-health/metrics/website/-/issues/40064.

[58] Tobias Höller. 2021. V3 onion services usage. (September 2021). Retrieved 07/26/2022 from https://blog.torproject.org/v3-onion-services-usage.

[59] Tobias Höller, Thomas Raab, Michael Roland, and René Mayrhofer. 2021. On the feasibility of short-lived dynamic onion services. In *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, San Francisco, CA, USA, (May 2021), pp. 25–30. DOI: 10.1109/SPW53761.2021.00012.

[60] Tobias Höller, Michael Roland, and René Mayrhofer. 2021. On the state of V3 onion services. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Free and Open Communications on the Internet (FOCI '21)*. ACM, Virtual, (August 2021), pp. 50–56. DOI: 10.1145/3473604.3474565.

[61] I2P. 2003. The Invisible Internet Project. Retrieved 07/28/2022 from https://geti2p.net/.

[62] Dark River Systems Inc. 2022. Hunchly Daily Dark Web Reports. Retrieved 09/06/2022 from https://www.hunch.ly/darkweb-osint/.

[63] International Organization for Standardization. 2021. Cards and security devices for personal identification — Building blocks for identity management via mobile devices. ISO/IEC DIS 23220-1. (November 2021). https://www.iso.org/standard/74910.html.

[64] International Organization for Standardization. 2016. Personal identification — ISO-compliant driving licence — Part 5: Mobile driving licence (mDL) application. en. Standard ISO/IEC TR 29110-1:2016. Geneva, CH. https://www.iso.org/standard/62711.html.

[65] Anja Jerichow, Jan Muller, Andreas Pfitzmann, Birgit Pfitzmann, and Michael Waidner. 1998. Real-time mixes: a bandwidth-efficient anonymity protocol. *IEEE Journal on Selected Areas in Communications*, 16, 4, (May 1998), 495–509. DOI: 10.1109/49.668973. https://doi.org/10.1109/49.668973.

[66] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. 2013. Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (*CCS '13*). Association for Computing Machinery, Berlin, Germany, pp. 337–348. ISBN: 9781450324779. DOI: 10.1145/2508859.2516651. https://doi.org/10.1145/2508859.2516651.

[67] Eric C Johnson. 1998. From the Inkpad to the Mousepad: IAFIS and Fingerprint Technology at the Dawn of the 21't Century. *Washington, DC: Bureau of Justice Assistance*.

[68] juga. 2019. How Bandwidth Scanners Monitor The Tor Network. Retrieved 07/29/2022 from https://blog.torproject.org/how-bandwidth-scanners-monitor-tor-network.

[69] George Kadianakis, David Goulet, and Roger Dingledine. 2020. A First Take at PoW Over Introduction Circuits. Retrieved 09/06/2022 from https://github.com/torproject/torspec/blob/main/proposals/327-pow-over-intro.txt.

[70] George Kadianakis, Mike Perry, David Goulet, and tevador. 2020. A First Take at PoW Over Introduction Circuits. (April 2020). Retrieved 08/30/2022 from https://gitlab.torproject.org/tpo/core/torspec/-/blob/main/proposals/327-pow-over-intro.txt.

[71] Achim Killer. 2021. Killer's Security: Aus dem Dunkel des Netzes. (December 2021). Retrieved 09/26/2022 from https://www.br.de/nachrichten/netzwelt/killer-s-security-aus-dem-dunkel-des-netzes,SrBF0h7.

[72] Stephan A. Kollmann and Alastair R. Beresford. 2017. The Cost of Push Notifications for Smartphones Using Tor Hidden Services. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 76–85. DOI: 10.1109/EuroSPW.2017.55.

[73] Georg Koppen. 2022. Keep rejecting relays/bridges with EOL version showing up. (March 2022). Retrieved 08/08/2022 from https://gitlab.torproject.org/tpo/network-health/team/-/issues/210.

[74] Georg Koppen. 2022. Malicious relays and the health of the Tor network. (April 2022). Retrieved 07/26/2022 from https://blog.torproject.org/malicious-relays-health-tor-network/.

[75] Michael Koster, Ari Keränen, and Jaime Jimenez. 2022. Publish-Subscribe Broker for the Constrained Application Protocol (CoAP). Internet-Draft draft-ietf-core-coap-pubsub-10. https://datatracker.ietf.org/doc/html/draft-ietf-core-coap-pubsub-10. Work in progress. Internet Engineering Task Force, (May 2022), 24 pages.

[76] Jan Koum and Brian Acton. [n. d.] WhatsApp. Retrieved 09/24/2022 from https://www.whatsapp.com.

[77] Stefan Krempl. 2021. Tor-Netzwerk: KAX17 führt massive Deanonymisierungsangriffe durch. (December 2021). Retrieved 09/26/2022 from https://www.heise.de/news/Tor-Netzwerk-KAX17-fuehrt-massive-Deanonymisierungsangriffe-durch-6286564.html.

[78] Bernd Kreuss. [n. d.] TorChat. Retrieved 09/24/2022 from https://github.com/prof7bit/TorChat.

[79] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. 2016. Riffle: An Efficient Communication System with Strong Anonymity. In *Proceedings on Privacy Enhancing Techonologies Symposium 2016*. Volume 2016. Walter de Gruyter GmbH, pp. 115–134. DOI: 10.1515/popets-2016-0008. https://petsymposium.org/popets/2016/popets-2016-0008.php.

[80] Micah Lee. 2014. OnionShare. Retrieved 08/06/2022 from https://onionshare.org/.

[81] Marcus Leech, Matt Ganis, Ying Da Lee, Ron Kurs, David Koblas, and LaMont Jones. 1996. SOCKS Protocol Version 5. RFC 1928. RFC Editor, (March 1996). https://datatracker.ietf.org/doc/html/rfc1928.

[82] Jörg Lenhard, Karsten Loesing, and Guido Wirtz. 2009. Performance Measurements of Tor Hidden Services in Low-Bandwidth Access Networks. In *Applied Cryptography and Network Security*. Springer Berlin Heidelberg, pp. 324–341. ISBN: 9783642019579.

[83] Sarah Jamie Lewis. 2018. Cwtch: Privacy Preserving Infrastructure for Asynchronous, Decentralized, Multi-Party and Metadata Resistant Applications. https://cwtch.im/cwtch.pdf.

[84] Sarah Jamie Lewis. 2016. The OnionScan Project. Retrieved 08/05/2022 from https://onionscan.org/.

[85] Ji Li, Chunxiang Gu, Xieli Zhang, Xi Chen, and Wenfen Liu. 2021. AttCorr: A Novel Deep Learning Model for Flow Correlation Attacks on Tor. In *2021 IEEE International Conference on Consumer Electronics and Computer Engineering (ICCECE)*, pp. 427–430. DOI: 10.1109/ICCECE51280.2021.9342534.

[86] Karsten Loesing, Steven J. Murdoch, and Roger Dingledine. 2010. A Case Study on Measuring Statistical Data in the Tor Anonymity Network. In *Proceedings of the Workshop on Ethics in Computer Security Research (WECSR 2010)* (*LNCS*). Springer, Tenerife, Canary Islands, Spain, (January 2010).

[87] Karsten Loesing, Werner Sandmann, Christian Wilms, and Guido Wirtz. 2008. Performance Measurements and Statistics of Tor Hidden Services. In *2008 International Symposium on Applications and the Internet*, pp. 1–7.

[88] Ewen Macaskill and Gabriel Dance. 2013. NSA Files: Decoded. (November 2013). Retrieved 07/26/2022 from https://www.theguardian.com/world/interactive/2013/nov/01/snowden-nsa-files-surveillance-revelations-decoded.

[89] Moxie Marlinspike. [n. d.] Signal. Retrieved 09/24/2022 from https://signal.org.

[90] Nick Mathewson. 2021. Announcing Arti, a pure-Rust Tor implementation. (July 2021). Retrieved 08/12/2022 from https://blog.torproject.org/announcing-arti/.

[91] René Mayrhofer, Michael Roland, and Tobias Höller. 2020. Poster: Towards an Architecture for Private Digital Authentication in the Physical World. In *Network and Distributed System Security Symposium (NDSS Symposium 2020), Posters*. San Diego, CA, USA, (February 2020).

[92] Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. 2009. Scalable onion routing with torsk. In *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 590–599.

[93] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. 2011. {PIR-Tor}: Scalable Anonymous Communication Using Private Information Retrieval. In *20th USENIX Security Symposium (USENIX Security 11)*.

[94] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. *The Cryptography Mailing List*, (October 2008). https://bitcoin.org/en/bitcoin-paper.

[95] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2018. DeepCorr: Strong Flow Correlation Attacks on Tor Using Deep Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (*CCS '18*). Association for Computing Machinery, Toronto, Canada, pp. 1962–1976. ISBN: 9781450356930. DOI: 10.1145/3243734.3243824. https://doi.org/10.1145/3243734.3243824.

[96] Iynkaran Natgunanathan, Abid Mehmood, Yong Xiang, Gleb Beliakov, and John Yearwood. 2016. Protection of privacy in biometric data. *IEEE access*, 4, 880–892.

[97] Juha Nurmi. 2014. Ahmia. Retrieved 09/06/2022 from https://ahmia.fi/.

[98] nusenu. 2020. How Malicious Tor Relays are Exploiting Users in 2020 (Part I). (August 2020). Retrieved 07/29/2022 from https://nusenu.medium.com/how-malicious-tor-relays-are-exploiting-users-in-2020-part-i-1097575c0cac.

[99] nusenu. 2021. Is "KAX17" performing de-anonymization Attacks against Tor Users? (November 2021). Retrieved 07/26/2022 from https://nusenu.medium.com/is-kax17-performing-de-anonymization-attacks-against-tor-users-42e566defce8.

[100] nusenu. 2021. say hi (and goodbye) to >1000 new exit relays at OVH. Retrieved 08/07/2022 from https://lists.torproject.org/pipermail/tor-relays/2021-May/019644.html.

[101] Open Privacy Research Society. [n. d.] cwtch. Retrieved 09/24/2022 from https://cwtch.im/.

[102] Gareth Owen and Nick Savage. 2016. Empirical analysis of Tor Hidden Services. *IET Information Security*, 10, 3, 113–118. DOI: 10.1049/iet-ifs.2015.0121.

[103] Pierluigi Paganini. 2021. KAX17 threat actor is attempting to deanonymize Tor users running thousands of rogue relays. (December 2021). Retrieved 09/26/2022 from https://securityaffairs.co/wordpress/125248/hacking/kax17-threat-actor-tor.html.

[104] Tom Phillips. 2020. Kazakhstan to roll out face recognition transit ticketing in 2021. (October 2020). Retrieved 07/27/2022 from https://www.nfcw.com/2020/10/20/368773/kazakhstan-to-roll-out-face-recognition-transit-ticketing-in-2021/.

[105] Tom Phillips. 2020. Moscow Metro to roll out biometric ticketing across entire network. (September 2020). Retrieved 07/27/2022 from https://www.nfcw.com/2020/09/10/367826/japanese-passengers-test-facial-recognition-ticketing-on-driverless-buses/.

[106] Tom Phillips. 2021. Moscow Metro to roll out biometric ticketing across entire network. (September 2021). Retrieved 07/27/2022 from https://www.nfcw.com/transit-ticketing-today/moscow-metro-to-roll-out-biometric-ticketing-across-entire-network/.

[107] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. 2017. The Loopix Anonymity System. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, (August 2017), pp. 1199–1216. ISBN: 978-1-931971-40-9. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/piotrowska.

[108] Khaira Rachna. 2018. Rs 500, 10 minutes, and you have access to billion Aadhaar details. Retrieved 07/26/2022 from https://www.tribuneindia.com/news/archive/nation/rs-500-10-minutes-and-you-have-access-to-billion-aadhaar-details-523361.

[109] Nalini K. Ratha, Jonathan H. Connell, and Ruud M. Bolle. 2001. Enhancing security and privacy in biometrics-based authentication systems. *IBM systems Journal*, 40, 3, 614–634.

[110] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. RFC Editor, (August 2018). https://datatracker.ietf.org/doc/html/rfc8446.

[111] Michael Rogers, Saitta Eleanor, Torsten Grote, Julian Dehm, Nico Alt, Benedikt Wieder, Ernir Erlingsson, Jack Grigg, and Bernard Tyers. [n. d.] Briar. Retrieved 09/24/2022 from https://briarproject.org/.

[112] sednet. 2013. TOR wants your spare bandwidth. Retrieved 08/07/2022 from https://www.linode.com/community/questions/8475/tor-wants-your-spare-bandwidth.

[113] David Shaw. 2003. The OpenPGP HTTP Keyserver Protocol (HKP). (March 2003). Retrieved 08/03/2022 from https://datatracker.ietf.org/doc/html/draft-shaw-openpgp-hkp-00.

[114] Ming Song, Gang Xiong, Zhenzhen Li, Junrui Peng, and Li Guo. 2013. A de-anonymize attack method based on traffic analysis. In *2013 8th International Conference on Communications and Networking in China (CHINACOM)*, pp. 455–460. DOI: 10.1109/ChinaCom.2013.6694639.

[115] Manu Sporny, Dave Longley, and David Chadwick. 2022. Verifiable Credentials Data Model. W3C Recommendation vc-data-model-20220303. W3C, (March 2022). https://www.w3.org/TR/2022/REC-vc-data-model-20220303/%22.

[116] Vudali Srinath. 2019. Aadhaar details of 7.82 crore from Telangana and Andhra found in possession of IT Grids (India) Pvt Ltd. Retrieved 07/26/2022 from https://timesofindia.indiatimes.com/city/hyderabad/aadhaar-details-of-7-82-crore-from-telangana-and-andhra-found-in-possession-of-it-grids-india-pvt-ltd/articleshow/68865938.cms.

[117] sukhbir. 2018. Sunsetting Tor Messenger. (April 2018). Retrieved 09/24/2022 from https://blog.torproject.org/sunsetting-tor-messenger/.

[118] sukhbir. 2015. Tor Messenger Beta: Chat over Tor, Easily. (October 2015). Retrieved 09/24/2022 from https://blog.torproject.org/tor-messenger-beta-chat-over-tor-easily/.

[119] The Thales Group. 2020. Automated Fingerprint Identification System (AFIS) overview - A short history. (January 2020). Retrieved 08/02/2022 from https://www.thalesgroup.com/en/markets/digital-identity-and-security/government/biometrics/afis-history.

[120] The Tor Project. 2008. Bridges. Retrieved 07/28/2022 from https://tb-manual.torproject.org/bridges/.

[121] The Tor Project. 2014. Research Safety Board. Retrieved 08/05/2022 from https://research.torproject.org/safetyboard.

[122] The Tor Project. 2004. TC: A Tor control protocol (Version 1). Retrieved 07/29/2022 from https://github.com/torproject/torspec/blob/main/control-spec.txt.

[123] The Tor Project. 2004. The Tor Project. Retrieved 07/28/2022 from.

[124] The Tor Project. 2005. Tor directory protocol, version 3. Retrieved 07/29/2022 from https://github.com/torproject/torspec/blob/main/dir-spec.txt.

[125] The Tor Project. 2021. Tor Metrics. Retrieved 07/29/2022 from https://metrics.torproject.org.

[126] The Tor Project. 2005. Tor Rendezvous Specification. Retrieved 07/28/2020 from https://github.com/torproject/torspec/blob/main/attic/rend-spec-v2.txt.

[127] The Tor Project. 2018. Tor Rendezvous Specification - Version 3. Retrieved 07/28/2022 from https://github.com/torproject/torspec/blob/main/rend-spec-v3.txt.

[128] Speek! UG. [n. d.] Speek. Retrieved 09/24/2022 from https://speek.network.

[129] Speek! UG. 2022. speek. Retrieved 08/07/2022 from https://speek.network.

[130] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles* (*SOSP '15*). Association for Computing Machinery, Monterey, California, pp. 137–152. ISBN: 9781450338349. DOI: 10.1145/2815400.2815417. https://doi.org/10.1145/2815400.2815417.

[131] youiopmop. 2019. Setup a Tor relay on FreeBSD 12. (2019). Retrieved 08/07/2022 from https://community.hetzner.com/tutorials/setup-a-tor-relay-on-freebsd-12.

Appendix A

Measure onion service creation times

This appendix extends the research presented in chapter 5 by explaining in detail how the information needed for the analysis of onion service deployment performance was obtained. An important remark to make here is that this research was conducted in the second half of 2020 when V2 onion services were not yet deprecated and the functionality of the Vanguards extension was not yet included in the regular Tor code. With the exception of V2 onion services, the measurement for V3 onion services, ephemeral V3 onion services and ephemeral V3 onion services with Vanguard should be reproducible with the information provided below. Pay attention however, that newer Tor releases have started to include parts of the Vanguard extension, so comparing onion services with and without Vanguard would require modification of the Tor code or using an older client.

To measure the times needed to create an onion service, timestamps for the following events were needed:

1. Circuits to introduction points are launched
2. Circuits to introduction points are established
3. Introduction points are ready
4. Creation of the main descriptor starts
5. Creation of the backup descriptor starts
6. Circuit to HSDir is launched (6x for V2 and 16x for V3)
7. Circuit to HSDir is established (6x for V2 and 16x for V3)
8. Upload to HSDir begins (6x for V2 and 16x for V3)
9. Upload to HSDir finishes (6x for V2 and 16x for V3)

Most of those events were tracked by subscribing to events already supported by Tor's control protocol [122]. We used the STEM[1] library for Python to subscribe to those events and extract timestamps from them. The only exception to this rule was the question when introduction points are ready. Listings A.1 and A.2 show the log statements we listened for to timestamp this event.

Listing A.1: Logging V2 *INTRO_ESTABLISHED* cells **rendservice.c**

```
3437 intro->circuit_established = 1;
3438 /* We might not have every introduction point ready but at this
3439 * point we know that the descriptor needs to be uploaded. */
3440 service->desc_is_dirty = time(NULL);
```

[1]https://stem.torproject.org/

```
3441 circuit_change_purpose(TO_CIRCUIT(circuit),
3442 CIRCUIT_PURPOSE_S_INTRO);
3443
3444 log_debug(LD_REND, "Received INTRO_ESTABLISHED cell on circuit"
3445 " %u (id: %" PRIu32 ") for service %s",
3446 (unsigned)circuit->base_.n_circ_id,
3447 circuit->global_identifier, serviceid);
3448
3449 /* Getting a valid INTRODUCE_ESTABLISHED means we've
3450 * successfully used the circ */
3451 pathbias_mark_use_success(circuit);
```

Listing A.2: Logging V3 *INTRO_ESTABLISHED* cells **hs_service.c**

```
3469 /* Update metrics. */
3470 hs_metrics_new_established_intro(service);
3471
3472 log_info(LD_REND, "Successfully received an INTRO_ESTABLISHED "
3473 "cell on circuit %u for service %s",
3474 TO_CIRCUIT(circ)->n_circ_id,
3475 safe_str_client(service->onion_address));
3476 return 0;
```

The remaining timestamps were collected by listening for these control protocol events:

- *CIRC* event[2]: Indicates that a circuit has changed. This happens every time a circuit is either launched or fully established. So this event alone provides all the information needed to find out when introduction point and HSDir upload circuits are both launched and ready.
- *HS_DESC* event[3]: Indicates that something happened with a hidden service descriptor. This event notifies us about the creation of new service descriptors, attempted uploads and successful uploads.

Combining the information provided by these two events with the timestamps for the log messages mentioned above, enabled us to collect all the information needed for the analysis of the onion service creation timings presented in chapter 5.

[2]https://stem.torproject.org/api/response.html#stem.response.events.CircuitEvent
[3]https://stem.torproject.org/api/response.html#stem.response.events.HSDescEvent

Appendix B

Improve Onion Service Latency

This appendix provides the source code of the modifications needed to achieve the improvements presented in chapter 7. For optimal readability, the presented code will always follow the same structure: An initial listing provides a diff with all the changes necessary to achieve the desired improvement. This listing omits complex sequences of code that were added (visualized by ...) as those are then provided in follow up listings with appropriate syntax highlighting. This provides readers with a good overview over the changes made as well as an easy to read source code when it comes to newly implemented features. Development branched out from Git commit 8ead53330c73e9bc1b82f6b7fc8946d629063842[1] and each of the presented improvements build upon the changes from previous sections.

B.1 Make the HSDir Optional

Making the HSDir optional requires two changes. Listing B.1 shows the location in the Tor source code where a change was necessary to emit additional *DESCRIPTOR_CONTENT* events and listing B.2 shows the source code that was added.

Listing B.1: Changes to notify about unpublished descriptors

```
diff --git a/src/feature/hs/hs_service.c b/src/feature/hs/hs_service.c
--- a/src/feature/hs/hs_service.c
+++ b/src/feature/hs/hs_service.c
@@ -62,6 +62,9 @@
#include "trunnel/hs/cell_common.h"
#include "trunnel/hs/cell_establish_intro.h"

+//TH: Added dependency here:
+#include "feature/control/control_events.h"
+
#ifdef HAVE_SYS_STAT_H
#include <sys/stat.h>
#endif

@@ -3008,6 +3011,16 @@ upload_descriptor_to_all(const hs_service_t *service,
tor_assert(service);
tor_assert(desc);

+  /* Check if the descriptor will be published. If it is not, trigger a
+   * HS_DESC_CONTENT event to enable users to use the service. */
```

[1]https://gitlab.torproject.org/tpo/core/tor/-/commit/8ead53330c73e9bc1b82f6b7fc8946d629063842

```
+  if (!get_options()->PublishHidServDescriptors) {
  …
+  }
+
```

Listing B.2: Emitting unpublished descriptors hs_service.c

```
3016 /* TH: Check if the descriptor will be published. If it is not,
3017 * trigger a HS_DESC_CONTENT event to enable users to use the
3018 * service. */
3019 if (!get_options()->PublishHidServDescriptors) {
3020   char *logged_desc = NULL;
3021   char base64_blinded_pk[ED25519_BASE64_LEN + 1];
3022   service_encode_descriptor(service, desc, &desc->signing_kp, &logged_desc);
3023   ed25519_public_to_base64(base64_blinded_pk,
      ↪  &desc->desc->plaintext_data.blinded_pubkey);
3024   control_event_hs_descriptor_content(service->onion_address, base64_blinded_pk,
      ↪  NULL,logged_desc);
3025 }
```

The second change required to bypass the HSDir implements the capability to load service descriptors directly via the control protocol. The changes necessary to add a new function to the control protocol are shown in listing B.3.

Listing B.3: Diff for loading a new service descriptor

```
diff --git a/src/feature/control/control_cmd.h b/src/feature/control/control_cmd.h
index 8cbe70a2ed..9ecb4db434 100644
--- a/src/feature/control/control_cmd.h
+++ b/src/feature/control/control_cmd.h
@@ -76,6 +76,7 @@ typedef struct control_cmd_syntax_t {
  #ifdef CONTROL_CMD_PRIVATE
  #include "feature/hs/hs_service.h"
+  #include "feature/hs/hs_cache.h"
  #include "lib/crypt_ops/crypto_ed25519.h"

  /* ADD_ONION secret key to create an ephemeral service. The command

@@ -2021,6 +2021,55 @@ handle_control_del_onion(control_connection_t *conn,
  return 0;
}
+  static const char *posthsdescriptor_keywords[] = {
+  "address", NULL,
+  …
+  }
+
static const control_cmd_syntax_t obsolete_syntax = {
  .max_args = UINT_MAX
};

@@ -2148,6 +2197,7 @@ static const control_cmd_def_t CONTROL_COMMANDS[] =
ONE_LINE(onion_client_auth_add, CMD_FL_WIPE),
ONE_LINE(onion_client_auth_remove, 0),
```

```
ONE_LINE(onion_client_auth_view, 0),
+  MULTLINE(posthsdescriptor, 0),
```

Listing B.4 shows the actual implementation of the new control protocol function. The initial definitions of *posthsdescriptor_keywords* and *posthsdescriptor_syntax* are used to define how the new control protocol function should be called and which arguments it should accept. In this case, the function accepts one named argument, the onion address of the service descriptor, and allows the client to append additional data lines to the request. This additional data has to contain the service descriptor that should be loaded. The function itself is very simple, it essentially calls one of Tor's internal functions (*hs_cache_store_as_client*) to store the service descriptor in Tor's internal cache.

Listing B.4: Function to load new descriptors **control_cmd.c**

```
2032 static const char *posthsdescriptor_keywords[] = {
2033   "address", NULL,
2034 };
2035
2036 static const control_cmd_syntax_t posthsdescriptor_syntax = {
2037   .max_args = 0,
2038   .accept_keywords = true,
2039   .allowed_keywords = posthsdescriptor_keywords,
2040   .want_cmddata = true,
2041 };
2042
2043 /**
2044 * Called when we receive a POSTHSDESCRIPTOR event. Accepts same arguments
2045 * as POSTDESCRIPTOR.
2046 *
2047 */
2048 static int
2049 handle_control_posthsdescriptor(control_connection_t *conn,
2050 const control_cmd_args_t *cmd_args) {
2051
2052   log_warn(LD_GENERAL, "Received newly created control port event for loading a
      ↪  hidden service descriptor with data %s", cmd_args->cmddata);
2053   const config_line_t *line;
2054   line = config_line_find_case(cmd_args->kwargs, "address");
2055   if (line) {
2056     log_warn(LD_GENERAL, "Successfully received encoded onion address argument: %s",
      ↪  line->value);
2057
2058   }else {
2059     //TODO Should the function also support loading descriptors for directory
      ↪  purposes?
2060     control_write_endreply(conn, 400, "No address provided, cannot decrypt
      ↪  descriptor");
2061     return -1;
2062   }
2063
2064   //TODO2: Need to convert that onion address in the right format.
2065   ed25519_public_key_t identity_pk;
2066   hs_parse_address(line->value, &identity_pk, NULL, NULL);
2067
```

```
2068   int status = hs_cache_store_as_client(cmd_args->cmddata, &identity_pk);
2069   log_warn(LD_GENERAL, "The attempt to load the descriptor led to: %i", status);
2070   if(status<0){
2071     control_write_endreply(conn, 400, "Failed to parse provided descriptor");
2072     return 1;
2073   }else {
2074     send_control_done(conn);
2075     return 0;
2076   }
2077 }
```

B.2 Bundle Information in the INTRO Cell

Bundling additional information in the *INTRODUCE1* cell required more changes, because support for extensions is only present in the protocol implementation, not in the functions that are used to build the actual messages. This can be seen in the diff for *hs_cell.c* where the line that statically sets the number of extensions to zero was removed by our changes. The changes presented in listing B.5 can be roughly split in three different categories:

1. Add an extension to *INTRODUCE1* cells
2. Send an *INTRODUCE1* cell via the control protocol
3. Handle extensions within *INTRODUCE1* cells.

Changes in header files indicate that new functionality had to be added to a file that was previously not needed there. This mostly occurred for the *control_cmd.h* file where new functionality was needed to support manual crafting of *INTRODUCE1* cells.

Listing B.5: Diff for extended introduction requests

```
diff --git a/src/feature/control/control_cmd.c b/src/feature/control/control_cmd.c
index 2d51ae1b84..46381c2875 100644
--- a/src/feature/control/control_cmd.c
+++ b/src/feature/control/control_cmd.c
@@ -33,6 +33,7 @@
#include "feature/control/control_getinfo.h"
#include "feature/control/control_proto.h"
#include "feature/hs/hs_control.h"
+#include "feature/hs/hs_client.h"
#include "feature/hs/hs_service.h"
#include "feature/nodelist/nodelist.h"
#include "feature/nodelist/routerinfo.h"
@@ -54,6 +55,13 @@

#include "app/config/statefile.h"

+#include "lib/crypt_ops/crypto_curve25519.h"
+#include "lib/crypt_ops/crypto_ed25519.h"
+#include "core/or/extend_info_st.h"
+#include "core/or/relay.h"
+#include "feature/hs/hs_cell.h"
+#include "core/or/crypt_path_st.h"
```

```
+
static int control_setconf_helper(control_connection_t *conn,
const control_cmd_args_t *args,
int use_defaults);
@@ -2073,1 +2073,211 @@ handle_control_posthsdescriptor(control_connection_t *conn,

+static const char *launch_by_extend_keywords[] = {
+  "onionaddress", NULL,
+  …
+}
+
+/*TH:  Helper Function for POSTHSSUB to create the cell extension fields. */
+static void
+cell_extension_set_field_to_string(trn_cell_extension_field_t *inp, const char*
↪  value, uint8_t type){
+  …
+}
+
+static const char *posthssub_keywords[] = {
+  "publisher", "circuit", "callback", "condition", NULL,
+  …
}

@@ -2198,5 +2420,7 @@ static const control_cmd_def_t CONTROL_COMMANDS[] =
ONE_LINE(onion_client_auth_remove, 0),
ONE_LINE(onion_client_auth_view, 0),
MULTLINE(posthsdescriptor, 0),
+  ONE_LINE(posthssub, 0),
+  ONE_LINE(launch_by_extend, 0),
};

diff --git a/src/feature/control/control_events.c
↪  b/src/feature/control/control_events.c
index e2aca6c03e..e740d32c09 100644
--- a/src/feature/control/control_events.c
+++ b/src/feature/control/control_events.c
@@ -109,6 +109,7 @@ const struct control_event_t control_event_table[] = {
{ EVENT_HS_DESC, "HS_DESC" },
{ EVENT_HS_DESC_CONTENT, "HS_DESC_CONTENT" },
{ EVENT_NETWORK_LIVENESS, "NETWORK_LIVENESS" },
+  { EVENT_HS_SUB, "HS_SUB"},
{ 0, NULL },
};

@@ -2249,6 +2250,14 @@ control_event_hs_descriptor_upload_failed(const char *id_digest,
id_digest, reason);
}

+/** Send HS_SUB event to inform controller about incoming subscription
+ * via one of our onion services
+ */
+void
+control_event_hs_sub(const char *condition, const char *callback){
+  send_control_event(EVENT_HS_SUB, "650 HS_SUB %s %s\r\n", condition, callback);
+}
+
void
control_events_free_all(void)
{
```

```
diff --git a/src/feature/control/control_events.h
↪  b/src/feature/control/control_events.h
index 68269cabba..5dcc5a13e2 100644
--- a/src/feature/control/control_events.h
+++ b/src/feature/control/control_events.h
@@ -222,6 +222,8 @@ void cbt_control_event_buildtimeout_set(const
↪  circuit_build_times_t *cbt,

int control_event_enter_controller_wait(void);

+void control_event_hs_sub(const char *condition, const char *callback);
+
void control_events_free_all(void);

#ifdef CONTROL_MODULE_PRIVATE
@@ -281,6 +283,7 @@ typedef uint64_t event_mask_t;
#define EVENT_PT_LOG                  0x0024
#define EVENT_PT_STATUS               0x0025
#define EVENT_MAX_                    0x0025
+#define EVENT_HS_SUB                    0x0026

/* sizeof(control_connection_t.event_mask) in bits, currently a uint64_t */
#define EVENT_CAPACITY_               0x0040

diff --git a/src/feature/hs/hs_cell.c b/src/feature/hs/hs_cell.c
index f84407de9e..152aa5fc6c 100644
--- a/src/feature/hs/hs_cell.c
+++ b/src/feature/hs/hs_cell.c
@@ -17,6 +17,8 @@

#include "core/or/origin_circuit_st.h"

+#include "feature/control/control_events.h"
+
/* Trunnel. */
#include "trunnel/ed25519_cert.h"
#include "trunnel/hs/cell_common.h"
@@ -372,6 +374,20 @@ introduce1_encrypt_and_encode(trn_cell_introduce1_t *cell,
tor_free(encrypted);
}

+/*TH:  Add function to add strings of any length into an extension field. */
+static void
+cell_extension_set_field_to_string(trn_cell_extension_field_t *inp, const char*
↪  value, uint8_t type){
+  …
+}
+
/** Using the INTRODUCE1 data, setup the ENCRYPTED section in cell. This means
* set it, encrypt it and encode it. */
static void
@@ -388,9 +404,23 @@ introduce1_set_encrypted(trn_cell_introduce1_t *cell,
tor_assert(enc_cell);

/* Set extension data. None are used. */
+  //TH: TODO Need to add functionality here to allow setting extensions.
ext = trn_cell_extension_new();
tor_assert(ext);
```

```
-  trn_cell_extension_set_num(ext, 0);
+  //TH: Create 3 fields for condition, callback and descriptor
+  trn_cell_extension_field_t *condition = trn_cell_extension_field_new();
+  trn_cell_extension_field_t *callback = trn_cell_extension_field_new();
+  trn_cell_extension_field_t *descriptor = trn_cell_extension_field_new();
+  //TH: Set values:
+  cell_extension_set_field_to_string(condition, "Face: 123123124123123123", 31);
+  cell_extension_set_field_to_string(callback, "blablablablabla.onion", 32);
+  cell_extension_set_field_to_string(descriptor, "here could be a service
↪  descriptor", 33);
+  //TH: Now set extension size and add to values
+  trn_cell_extension_set_num(ext, 3);
+  trn_cell_extension_setlen_fields(ext, 3);
+  trn_cell_extension_set_fields(ext, 0, condition);
+  trn_cell_extension_set_fields(ext, 1, callback);
+  trn_cell_extension_set_fields(ext, 2, descriptor);
trn_cell_introduce_encrypted_set_extensions(enc_cell, ext);

/* Set the rendezvous cookie. */
@@ -413,6 +443,43 @@ introduce1_set_encrypted(trn_cell_introduce1_t *cell,
trn_cell_introduce_encrypted_free(enc_cell);
}

+
+introduce1_set_extended_encrypted(trn_cell_introduce1_t *cell,
+                          const hs_cell_introduce1_data_t *data,
+                          trn_cell_extension_t *private_extensions)
+{
+  …
+}
+
+
/** Set the authentication key in the INTRODUCE1 cell from the given data. */
static void
introduce1_set_auth_key(trn_cell_introduce1_t *cell,
@@ -863,6 +930,45 @@ hs_cell_parse_introduce2(hs_cell_introduce2_data_t *data,
}

/* XXX: Implement client authorization checks. */
+  //TH: First draft of my own extension handling.
+ …
+

/* Extract onion key and rendezvous cookie from the cell used for the
* rendezvous point circuit e2e encryption. */
@@ -975,6 +1081,46 @@ hs_cell_build_introduce1(const hs_cell_introduce1_data_t *data,
return cell_len;
}

+/** Build an INTRODUCE1 cell with arbitrary extensions from the given data.
+ * The encoded cell is put in cell_out which must be of at least size
+ * RELAY_PAYLOAD_SIZE. On success, the encoded length is returned else a
+ * negative value and the content of cell_out should be ignored. */
+ssize_t
+hs_cell_build_extended_introduce1(const hs_cell_introduce1_data_t *data,
+                          trn_cell_extension_t *public_extensions,
+                          trn_cell_extension_t *private_extensions,
+                          uint8_t *cell_out)
+{
```

```
+  …
+}
+
+
+
/** Build an ESTABLISH_RENDEZVOUS cell from the given rendezvous_cookie. The
* encoded cell is put in cell_out which must be of at least
* RELAY_PAYLOAD_SIZE. On success, the encoded length is returned and the
diff --git a/src/feature/hs/hs_cell.h b/src/feature/hs/hs_cell.h
index dc083ca03f..5be2c2e656 100644
--- a/src/feature/hs/hs_cell.h
+++ b/src/feature/hs/hs_cell.h
@@ -11,6 +11,7 @@

#include "core/or/or.h"
#include "feature/hs/hs_service.h"
+#include "trunnel/hs/cell_common.h"

/** An INTRODUCE1 cell requires at least this amount of bytes (see section
* 3.2.2 of the specification). Below this value, the cell must be padded. */
@@ -96,6 +97,10 @@ ssize_t hs_cell_build_rendezvous1(const uint8_t *rendezvous_cookie,
uint8_t *cell_out);
ssize_t hs_cell_build_introduce1(const hs_cell_introduce1_data_t *data,
uint8_t *cell_out);
+ssize_t hs_cell_build_extended_introduce1(const hs_cell_introduce1_data_t *data,
+                          trn_cell_extension_t *public_extensions,
+                          trn_cell_extension_t *private_extensions,
+                          uint8_t *cell_out);
ssize_t hs_cell_build_establish_rendezvous(const uint8_t *rendezvous_cookie,
uint8_t *cell_out);
```

Listing B.6 shows the first of the two new functions that were added to the control protocol to support manual transmission of *INTRODUCE1* cells. Its structure is similar to the function shown in listing B.4 because it is again a new control protocol function. It is used to launch new circuits directly. As discussed in section 7.3.4, this function was added for convenience because the control protocol already has the capability to create new circuits. This function made testing our implementation easier, but is unlikely to be included in a stable implementation of this improvement.

Listing B.6: Launching introduction circuits **control_cmd.c**

```
2083 static const char *launch_by_extend_keywords[] = {
2084   "onionaddress", NULL,
2085 };
2086
2087 static const control_cmd_syntax_t launch_by_extend_syntax = {
2088   .max_args=0,
2089   .accept_keywords= true,
2090   .allowed_keywords = launch_by_extend_keywords,
2091   .want_cmddata = false,
2092 };
2093
2094 static int handle_control_launch_by_extend(control_connection_t *conn,
2095 const control_cmd_args_t *cmd_args){
2096
```

```
  const config_line_t *arg;
  for (arg = cmd_args->kwargs; arg; arg = arg->next) {
    log_info(LD_GENERAL, "THDEBUG: Received keyword argument with key %s and value
↪  %s", arg->key, arg->value);
  }
  const config_line_t *destination=config_line_find_case(cmd_args->kwargs,
↪  "onionaddress");
  if (destination){
    //Setting up the flags
    int flags = CIRCLAUNCH_NEED_CAPACITY;
    //if (want_onehop) flags |= CIRCLAUNCH_ONEHOP_TUNNEL;
    int need_internal=1;
    int need_uptime=0;
    if (need_uptime) flags |= CIRCLAUNCH_NEED_UPTIME;
    if (need_internal) flags |= CIRCLAUNCH_IS_INTERNAL;
    log_info(LD_GENERAL, "Flags have been set up to %i", flags);
    //Setting up the Purpose
    uint8_t new_circ_purpose = CIRCUIT_PURPOSE_C_INTRODUCING;
    ed25519_public_key_t identity_key;
    hs_parse_address(destination->value, &identity_key, NULL, NULL);
    hs_ident_edge_conn_t *hs_ident= hs_ident_edge_conn_new(&identity_key);
    edge_connection_t edge_conn;
    edge_conn.hs_ident = hs_ident;
    extend_info_t *extend_info = hs_client_get_random_intro_from_edge(&edge_conn);

    if(!extend_info){
      log_info(LD_GENERAL, "Failed to get introduction point");
    }
    log_info(LD_GENERAL, "Ready to launch introduction circuit");
    origin_circuit_t *circ = circuit_launch_by_extend_info(new_circ_purpose,
↪  extend_info, flags);
    circ->hs_ident=hs_ident_circuit_new(&identity_key);
    log_info(LD_GENERAL, "intro circuit (%i) has been launched",
↪  circ->global_identifier);
    control_printf_midreply(conn, 250, "%i", circ->global_identifier);
    send_control_done(conn);
    return 0;
  }else{
    control_write_endreply(conn, 400, "No valid destination onion address was
↪  provided");
    return 1;
  }
}
```

There is only one documented use of extensions within the current Tor implementation (DDoS protection at the introduction point), but it seems like Tor usually defines new extension types for stable extensions and has therefore no code that would easily allow users to use the generic extension format. Listing B.7 shows the function we implemented for this purpose. We do not expect this function to be part of a stable implementation of this improvement, it purely serves as a helper to lower the development effort required for this prototype.

Listing B.7: Defining a string extension control_cmd.c

```
/*TH:  Helper Function for POSTHSSUB to create the cell extension fields. */
static void
cell_extension_set_field_to_string(trn_cell_extension_field_t *inp, const char*
↪  value, uint8_t type){
  size_t string_length;
  string_length=strlen(value);
  trn_cell_extension_field_set_field_type(inp, type);
  trn_cell_extension_field_set_field_len(inp, (uint64_t) string_length);
  trn_cell_extension_field_setlen_field(inp, string_length);
  for (unsigned long i=0;i<string_length;i++) {
    char c = value[i];
    trn_cell_extension_field_set_field(inp, (size_t) i, (uint8_t)c);
  }
}
```

Just like listing B.6, listing B.8 specifies a new control protocol function. This function is used to send an *INTRODUCE1* cell directly via a circuit. It accepts four named arguments: The onion address of the receiver of the message, the circuit where the cell should be sent, the callback address that should be notified about events, and the condition when events should be sent. Note that this function does not yet support arbitrary extension data, at this point in development the prototype only supported static extension content.

Listing B.8: Send extended intro request control_cmd.c

```
static const char *posthssub_keywords[] = {
  "publisher", "circuit", "callback", "condition", NULL,
};

static const control_cmd_syntax_t posthssub_syntax = {
  .max_args = 0,
  .accept_keywords = true,
  .allowed_keywords = posthssub_keywords,
  .want_cmddata = false,
};

/**
* Called when we receive a POSTHSSUB request
* Accepts two arguments:
* "publisher" -> The onion address of the service to subscribe to
* "circuit" -> The circuit to use for creating the service
* "callback" -> The onion address to send as callback source
* "condition" -> The condition expected to trigger the callback
*/
static int
handle_control_posthssub(control_connection_t *conn,
const control_cmd_args_t *cmd_args)
{
  //First read arguments:
  const config_line_t *arg;

  for (arg = cmd_args->kwargs; arg; arg = arg->next) {
```

```
2177    log_info(LD_GENERAL, "Received keyword argument with key %s and value %s",
        ↪ arg->key, arg->value);
2178  }
2179  log_info(LD_GENERAL, "Finished receiving keywords");
2180  const config_line_t *publisher,*circuit,*callback,*condition;
2181  publisher = config_line_find_case(cmd_args->kwargs, "publisher");
2182  circuit = config_line_find_case(cmd_args->kwargs, "circuit");
2183  callback = config_line_find_case(cmd_args->kwargs, "callback");
2184  condition = config_line_find_case(cmd_args->kwargs, "condition");
2185  log_info(LD_GENERAL, "Received command to sub at %s requesting event info for '%s'
        ↪ to: %s", publisher->value, condition->value, callback->value);
2186
2187  ed25519_public_key_t identity_key;
2188  hs_parse_address(publisher->value, &identity_key, NULL, NULL);
2189  //First, we need the correct circuit to send the intro request for:
2190  origin_circuit_t *circ = get_circ(circuit->value);
2191  if (circ->hs_ident == NULL) {
2192    log_err(LD_GENERAL, "Error, no hs_ident on circuit");
2193  }
2194  if (circ!=NULL && circ->base_.state == CIRCUIT_STATE_OPEN){
2195    log_info(LD_GENERAL, "Circuit open, Start building introduction data");
2196    const hs_descriptor_t *descriptor= hs_cache_lookup_as_client(&identity_key);
2197    const hs_desc_intro_point_t *intro_point=NULL;
2198    log_info(LD_GENERAL, "Descriptor ready, starting loop");
2199    SMARTLIST_FOREACH_BEGIN(descriptor->encrypted_data.intro_points,
2200    const hs_desc_intro_point_t *, ip) {
2201      log_info(LD_GENERAL, "Entering loop");
2202      if (ed25519_pubkey_eq(&circ->hs_ident->intro_auth_pk,
2203      &ip->auth_key_cert->signed_key)) {
2204        log_info(LD_GENERAL, "Found introduction point!");
2205        intro_point= ip;
2206        break;
2207      }
2208    } SMARTLIST_FOREACH_END(ip);
2209    log_info(LD_GENERAL, "Introduction Point found");
2210    if (intro_point == NULL) {
2211      control_write_endreply(conn, 400, "Could not find out which IP we are using");
2212      return 1;
2213    }
2214
2215    curve25519_keypair_t rend_pubkey;
2216    uint8_t rend_cookie[REND_COOKIE_LEN];
2217    for (int i=0;i<REND_COOKIE_LEN;i++){
2218      uint8_t value=222;
2219      rend_cookie[i]=value;
2220    }
2221    curve25519_keypair_generate(&rend_pubkey, 0);
2222
2223    //create my own introduce1_data
2224    hs_cell_introduce1_data_t intro1_data;
2225    intro1_data.auth_pk = &intro_point->auth_key_cert->signed_key;
2226    intro1_data.enc_pk = &intro_point->enc_key;
2227    intro1_data.subcredential = &descriptor->subcredential;
2228    intro1_data.link_specifiers = intro_point->link_specifiers;
2229    intro1_data.onion_pk = &intro_point->onion_key;
2230    intro1_data.rendezvous_cookie=rend_cookie;
2231    intro1_data.client_kp=&rend_pubkey;
2232
2233    // Create extension data to send along
```

```
2234    log_info(LD_GENERAL, "Start building extensions to send along!");
2235    trn_cell_extension_t *public_extensions=trn_cell_extension_new();
2236    trn_cell_extension_t *private_extensions=trn_cell_extension_new();
2237    trn_cell_extension_field_t *condition_ext = trn_cell_extension_field_new();
2238    trn_cell_extension_field_t *callback_ext = trn_cell_extension_field_new();
2239    trn_cell_extension_field_t *callback_data_ext = trn_cell_extension_field_new();
2240    //TH: Set values:
2241    cell_extension_set_field_to_string(condition_ext, condition->value, 31);
2242    cell_extension_set_field_to_string(callback_ext, callback->value, 32);
2243    cell_extension_set_field_to_string(callback_data_ext, "TODO: Insert long
      descriptor here!", 33);
2244    //TH: Now set extension size and add to values
2245    trn_cell_extension_set_num(private_extensions, 3);
2246    trn_cell_extension_setlen_fields(private_extensions, 3);
2247    trn_cell_extension_set_fields(private_extensions, 0, condition_ext);
2248    trn_cell_extension_set_fields(private_extensions, 1, callback_ext);
2249    trn_cell_extension_set_fields(private_extensions, 2, callback_data_ext);
2250
2251    uint8_t cell_out[RELAY_PAYLOAD_SIZE]={0};
2252    log_info(LD_GENERAL, "Building introduce1 cell");
2253    ssize_t length=hs_cell_build_extended_introduce1(&intro1_data,
      public_extensions, private_extensions, cell_out);
2254
2255    if(length<0){
2256      log_info(LD_GENERAL, "THDEBUG: Error, failed to build extended introduce1
      cell!");
2257    }else{
2258      int returnCode=relay_send_command_from_edge(CONTROL_CELL_ID, TO_CIRCUIT(circ),
2259      RELAY_COMMAND_INTRODUCE1,
2260      (const char*) cell_out, length, circ->cpath->prev);
2261      if (returnCode) {
2262        log_warn(LD_GENERAL, "TH: Unable to send introduce cell on circuit %u",
      TO_CIRCUIT(circ)->n_circ_id);
2263      } else{
2264        circuit_change_purpose(TO_CIRCUIT(circ),
      CIRCUIT_PURPOSE_C_INTRODUCE_ACK_WAIT);
2265      }
2266    }
2267
2268    //int status = hs_client_send_introduce1(circ, &rend_circ_mock);
2269    log_info(LD_GENERAL, "Intro1 cell should have been sent");
2270    send_control_done(conn);
2271    return 0;
2272  }else{
2273    control_write_endreply(conn, 400, "Circuit is not yet ready for connection");
2274    return -1;
2275  }
2276 }
```

Listings B.9 and B.11 show the new functions needed to add extensions to the *INTRODUCE1* cell. This is necessary because the current implementation of the Tor client assumes extensions to always be empty, so functions have no optional argument via which extensions could be passed. This forced us to either change the method signatures of core parts of the Tor application or duplicate functions. To keep the changes made to the Tor client down to a minimum, we opted to duplicate the functions with an additional argument for extensions. The function *introduce1_set_extended_encrypted* is a duplication of *introduce1_set_encrypted* with an additional argument for potential extensions.

Listing B.9: Support extensions in introduction cells hs_cell.c

```
introduce1_set_extended_encrypted(trn_cell_introduce1_t *cell,
const hs_cell_introduce1_data_t *data,
trn_cell_extension_t *private_extensions)
{
  trn_cell_introduce_encrypted_t *enc_cell;
  trn_cell_extension_t *ext;

  tor_assert(cell);
  tor_assert(data);

  enc_cell = trn_cell_introduce_encrypted_new();
  tor_assert(enc_cell);

  trn_cell_introduce_encrypted_set_extensions(enc_cell, private_extensions);

  /* Set the rendezvous cookie. */
  memcpy(trn_cell_introduce_encrypted_getarray_rend_cookie(enc_cell),
  data->rendezvous_cookie, REND_COOKIE_LEN);

  /* Set the onion public key. */
  introduce1_set_encrypted_onion_key(enc_cell, data->onion_pk->public_key);

  /* Set the link specifiers. */
  introduce1_set_encrypted_link_spec(enc_cell, data->link_specifiers);

  /* Set padding. */
  introduce1_set_encrypted_padding(cell, enc_cell);

  /* Encrypt and encode it in the cell. */
  introduce1_encrypt_and_encode(cell, enc_cell, data);

  /* Cleanup. */
  trn_cell_introduce_encrypted_free(enc_cell);
}
```

Listing B.10 shows how to handle an *INTRODUCE1* cell with extensions. This requires iterating through the included extensions, extracting their values and deciding what to do with them. If all the necessary extensions are present, the newly added *HS_SUB* event is triggered to inform the onion service about the received subscription.

Listing B.10: Handle received extensions hs_cell.c

```
trn_cell_extension_t *extensions=
↪  trn_cell_introduce_encrypted_get_extensions(enc_cell);
uint8_t count=trn_cell_extension_get_num(extensions);
if (count > 0){
  char *condition,*callback;
  for(size_t i=0; i<count; i++) {
    trn_cell_extension_field_t *field = trn_cell_extension_get_fields(extensions, i);
    uint8_t type=trn_cell_extension_field_get_field_type(field);
    uint64_t length=trn_cell_extension_field_get_field_len(field);
    char content[length+1];
```

```
943     for(size_t j=0; j<length; j++) {
944       char c = (char) trn_cell_extension_field_get_field(field, j);
945       content[j]=c;
946     }
947     content[length]='\0';
948     log_debug(LD_GENERAL, "THDEBUG: Detected Extension with Type %i and %lu bytes of
    ↪  data: %s", type, length, content);
949     switch(type){
950       case 31:
951       condition=calloc(length+1, sizeof(char));
952       strcpy(condition, content);
953       break;
954       case 32:
955       callback=calloc(length+1, sizeof(char));
956       strcpy(callback, content);
957       break;
958       case 33:break; //TODO: Implement logic to load descriptor if descriptor is
    ↪  present. Need to verify size restrictions first
959       default: log_warn(LD_GENERAL, "TH: Detected unknown Extension with Type %i and
    ↪  %lu bytes of data: %s", type, length, content);
960     }
961   }
962   //TH: Now send the descriptor event, if possible
963   if (condition && callback){
964     log_info(LD_GENERAL, "TH: Received an HSSUB-Extension");
965     control_event_hs_sub(condition, callback);
966     free(condition);
967     free(callback);
968     goto done;
969   }
```

Listing B.11: Build extended intro request **hs_cell.c**

```
1084 ** Build an INTRODUCE1 cell with arbitrary extensions from the given data.
1085 * The encoded cell is put in cell_out which must be of at least size
1086 * RELAY_PAYLOAD_SIZE. On success, the encoded length is returned else a
1087 * negative value and the content of cell_out should be ignored. */
1088 ssize_t
1089 hs_cell_build_extended_introduce1(const hs_cell_introduce1_data_t *data,
1090 trn_cell_extension_t *public_extensions,
1091 trn_cell_extension_t *private_extensions,
1092 uint8_t *cell_out)
1093 {
1094   ssize_t cell_len;
1095   trn_cell_introduce1_t *cell;
1096   trn_cell_extension_t *ext;
1097
1098   tor_assert(data);
1099   tor_assert(cell_out);
1100
1101   cell = trn_cell_introduce1_new();
1102   tor_assert(cell);
1103
1104   /* Set extension data.*/
1105   //TODO: This assumes that extensions is not empty, should probably be checked
1106   trn_cell_introduce1_set_extensions(cell, public_extensions);
```

```
1107
1108   /* Set the authentication key. */
1109   introduce1_set_auth_key(cell, data);
1110
1111   /* Set the encrypted section. This will set, encrypt and encode the
1112   * ENCRYPTED section in the cell. After this, we'll be ready to encode. */
1113   introduce1_set_extended_encrypted(cell, data, private_extensions);
1114
1115   /* Final encoding. */
1116   cell_len = trn_cell_introduce1_encode(cell_out, RELAY_PAYLOAD_SIZE, cell);
1117
1118   trn_cell_introduce1_free(cell);
1119   return cell_len;
1120 }
```

B.3 Use Minimized Descriptors

The third improvement requires two significant changes. First, the service descriptor information has to be included in the *INTRODUCE1* cell, meaning that several new extension fields have to be added. Second, the capability to construct a descriptor from the information included in the *INTRODUCE1* cell must be implemented. This change is quite extensive, because Tor does not expect to load unencrypted descriptors as there has not been a use case for this before. Listing B.12 provides an overview over both of these changes.

Listing B.12: Diff for minimized descriptors

```
diff --git a/src/feature/control/control_cmd.c b/src/feature/control/control_cmd.c
index 46381c2875..8ca564855d 100644
--- a/src/feature/control/control_cmd.c
+++ b/src/feature/control/control_cmd.c
@@ -2050,10 +2050,6 @@ handle_control_posthsdescriptor(control_connection_t *conn,
 ↪  const control_cmd_args_t *cmd_args) {

   log_warn(LD_GENERAL, "Received newly created control port event for loading a hidden
 ↪  service descriptor with data %s", cmd_args->cmddata);
-  //Verify if the command line arguments work they way they are expected to.
-
-  // Try to load the provided descriptor into the cache.
-  //TODO1: Need to also provide the onion address (= identity key) so descriptor can
 ↪  be decrypted
   const config_line_t *line;
   line = config_line_find_case(cmd_args->kwargs, "address");
   if (line) {
@@ -2124,6 +2120,7 @@ static int handle_control_launch_by_extend(control_connection_t
 ↪  *conn,

   origin_circuit_t *circ = circuit_launch_by_extend_info(new_circ_purpose,
 ↪  extend_info, flags);
     circ->hs_ident=hs_ident_circuit_new(&identity_key);
     log_info(LD_GENERAL, "intro circuit (%i) has been launched",
 ↪  circ->global_identifier);
+    control_printf_midreply(conn, 250, "%i", circ->global_identifier);
```

```
     send_control_done(conn);
     return 0;
   }else{
@@ -2132,7 +2129,6 @@ static int handle_control_launch_by_extend(control_connection_t
↪  *conn,
   }
 }

-
 /*TH:  Helper Function for POSTHSSUB to create the cell extension fields. */
 static void
 cell_extension_set_field_to_string(trn_cell_extension_field_t *inp, const char*
↪  value, uint8_t type){
@@ -2146,6 +2142,16 @@ cell_extension_set_field_to_string(trn_cell_extension_field_t
↪  *inp, const char*
     trn_cell_extension_field_set_field(inp, (size_t) i, (uint8_t)c);
   }
 }
+static void
+cell_extension_set_field_to_bytes(trn_cell_extension_field_t *inp, uint64_t length,
↪  const uint8_t* data, uint8_t type){
+  trn_cell_extension_field_set_field_type(inp, type);
+  trn_cell_extension_field_set_field_len(inp, length );
+  trn_cell_extension_field_setlen_field(inp, length);
+  for (unsigned long i=0;i<length;i++) {
+    uint8_t byte = data[i];
+    trn_cell_extension_field_set_field(inp, (size_t) i, byte);
+  }
+}

 static const char *posthssub_keywords[] = {
   "publisher", "circuit", "callback", "condition", NULL,
@@ -2155,7 +2161,7 @@ static const control_cmd_syntax_t posthssub_syntax = {
   .max_args = 0,
   .accept_keywords = true,
   .allowed_keywords = posthssub_keywords,
-  .want_cmddata = false,
+  .want_cmddata = true,
 };

 /**
@@ -2165,6 +2171,7 @@ static const control_cmd_syntax_t posthssub_syntax = {
  * "circuit" -> The circuit to use for creating the service
  * "callback" -> The onion address to send as callback source
  * "condition" -> The condition expected to trigger the callback
+ * "duration" -> A UTC timestamp how long you want to sub (default=1hour)
  */
 static int
 handle_control_posthssub(control_connection_t *conn,
@@ -2177,58 +2184,39 @@ handle_control_posthssub(control_connection_t *conn,
     log_info(LD_GENERAL, "Received keyword argument with key %s and value %s",
↪  arg->key, arg->value);
   }
   log_info(LD_GENERAL, "Finished receiving keywords");
-  const config_line_t *publisher,*circuit,*callback,*condition;
+  const config_line_t *publisher,*circuit,*callback,*condition, *duration;
   publisher = config_line_find_case(cmd_args->kwargs, "publisher");
   circuit = config_line_find_case(cmd_args->kwargs, "circuit");
   callback = config_line_find_case(cmd_args->kwargs, "callback");
```

```
   condition = config_line_find_case(cmd_args->kwargs, "condition");
+  duration = config_line_find_case(cmd_args->kwargs, "duration");
   log_info(LD_GENERAL, "Received command to sub at %s requesting event info for '%s'
↪   to: %s", publisher->value, condition->value, callback->value);

   ed25519_public_key_t identity_key;
   hs_parse_address(publisher->value, &identity_key, NULL, NULL);
   //First, we need the correct circuit to send the intro request for:
   origin_circuit_t *circ = get_circ(circuit->value);
-  if (circ->hs_ident == NULL) {
+  if (circ == NULL || circ->hs_ident == NULL) {
     log_err(LD_GENERAL, "Error, no hs_ident on circuit");
+    control_write_endreply(conn, 400, "Error: No HS_IDENT set on provided circuit.
↪   Cannot send HSSUB");
+    return -1;
   }
   if (circ!=NULL && circ->base_.state == CIRCUIT_STATE_OPEN){
-    log_info(LD_GENERAL, "Circuit open, Start building introduction data");
     const hs_descriptor_t *descriptor= hs_cache_lookup_as_client(&identity_key);
     const hs_desc_intro_point_t *intro_point=NULL;
-    log_info(LD_GENERAL, "Descriptor ready, starting loop");
     SMARTLIST_FOREACH_BEGIN(descriptor->encrypted_data.intro_points,
                             const hs_desc_intro_point_t *, ip) {
-      log_info(LD_GENERAL, "Entering loop");
       if (ed25519_pubkey_eq(&circ->hs_ident->intro_auth_pk,
                             &ip->auth_key_cert->signed_key)) {
-        log_info(LD_GENERAL, "Found introduction point!");
         intro_point= ip;
         break;
       }
     } SMARTLIST_FOREACH_END(ip);
-    log_info(LD_GENERAL, "Introduction Point found");
     if (intro_point == NULL) {
-      control_write_endreply(conn, 400, "Could not find out which IP we are using");
+      control_write_endreply(conn, 400, "Could not find out which IntroPoint we are
↪   using");
       return 1;
     }
-     //SMARTLIST_FOREACH_BEGIN(descriptor->encrypted_data.intro_points,
↪   hs_desc_intro_point_t*, ip) {
-
-      //if( memcmp(&ip->onion_key, &extend_info->curve25519_onion_key,
↪   sizeof(curve25519_public_key_t) ) ) {
-      //  log_info(LD_GENERAL, "Found correct introduction point");
-        //ed25519_pubkey_copy(&circ->hs_ident->intro_auth_pk,
↪   &ip->auth_key_cert->signed_key);
-        //ip_enc_key=&ip->enc_key;
-      //}else{
-      //  log_info(LD_GENERAL, "Not a correct key");
-      //}
-    //} SMARTLIST_FOREACH_END(ip);
-
-    //Get Prepare the mocked rend_circuit
-    //origin_circuit_t rend_circ_mock;
-    //hs_ident_circuit_t hs_ident_mock;
-    //rend_circ_mock.hs_ident=&hs_ident_mock;
-    //cpath_build_state_t build_state_mock;
-    //build_state_mock.chosen_exit=extend_info;
-    //rend_circ_mock.build_state=&build_state_mock;
```

```
+
     curve25519_keypair_t rend_pubkey;
     uint8_t rend_cookie[REND_COOKIE_LEN];
     for (int i=0;i<REND_COOKIE_LEN;i++){
@@ -2237,52 +2225,121 @@ handle_control_posthssub(control_connection_t *conn,
     }
     curve25519_keypair_generate(&rend_pubkey, 0);

-    //create my own introduce1_data
     hs_cell_introduce1_data_t intro1_data;
     intro1_data.auth_pk = &intro_point->auth_key_cert->signed_key;
     intro1_data.enc_pk = &intro_point->enc_key;
     intro1_data.subcredential = &descriptor->subcredential;
-    intro1_data.link_specifiers = intro_point->link_specifiers;
     intro1_data.onion_pk = &intro_point->onion_key;
-    intro1_data.rendezvous_cookie=rend_cookie;
-    intro1_data.client_kp=&rend_pubkey;

-    // Create extension data to send along
-    log_info(LD_GENERAL, "Start building extensions to send along!");
     trn_cell_extension_t *public_extensions=trn_cell_extension_new();
     trn_cell_extension_t *private_extensions=trn_cell_extension_new();
+
+    …
+
+    // Create extension data to send along
+    log_debug(LD_GENERAL, "Start building extensions to send along!");
     trn_cell_extension_field_t *condition_ext = trn_cell_extension_field_new();
     trn_cell_extension_field_t *callback_ext = trn_cell_extension_field_new();
-    trn_cell_extension_field_t *callback_data_ext = trn_cell_extension_field_new();
+    trn_cell_extension_field_t *duration_ext = trn_cell_extension_field_new();
     //TH: Set values:
     cell_extension_set_field_to_string(condition_ext, condition->value, 31);
     cell_extension_set_field_to_string(callback_ext, callback->value, 32);
-    cell_extension_set_field_to_string(callback_data_ext, "TODO: Insert long
↪  descriptor here!", 33);
+    if(duration){
+      cell_extension_set_field_to_string(duration_ext, duration->value, 33);
+    }else{
+      char* default_duration=tor_calloc(22, sizeof(char));
+      log_debug(LD_GENERAL, "Got until here");
+      log_debug(LD_GENERAL, "Time is: %li", time(NULL));
+      log_debug(LD_GENERAL, "Expiry deadline is: %li + %li", time(NULL), 7200l);
+      time_t expiry_time=time(NULL) + (time_t)7200;
+      log_debug(LD_GENERAL, "Expiry deadline is: %li + %li = %li", time(NULL), 7200l,
↪  expiry_time);
+      snprintf(default_duration,(size_t) 22, "%li", time(NULL) + 7200l);
+      cell_extension_set_field_to_string(duration_ext, default_duration, 33);
+      free(default_duration);
+    }
     //TH: Now set extension size and add to values
-    trn_cell_extension_set_num(private_extensions, 3);
-    trn_cell_extension_setlen_fields(private_extensions, 3);
     trn_cell_extension_set_fields(private_extensions, 0, condition_ext);
     trn_cell_extension_set_fields(private_extensions, 1, callback_ext);
-    trn_cell_extension_set_fields(private_extensions, 2, callback_data_ext);
+    trn_cell_extension_set_fields(private_extensions, 2, duration_ext);

     uint8_t cell_out[RELAY_PAYLOAD_SIZE]={0};
```

```
     log_info(LD_GENERAL, "Building introduce1 cell");
     ssize_t length=hs_cell_build_extended_introduce1(&intro1_data, public_extensions,
↪  private_extensions, cell_out);

     if(length<0){
-      log_info(LD_GENERAL, "THDEBUG: Error, failed to build extended introduce1
↪  cell!");
+      log_info(LD_GENERAL, "TH: Error, failed to build extended introduce1 cell!");
+      control_write_endreply(conn, 400, "Error: Error occurred when building extended
↪  introduce1 cell");
+      return -1;
     }else{
+      log_info(LD_GENERAL, "TH: Sending introduce cell of size %li (of %i
↪  available)",length, RELAY_PAYLOAD_SIZE);
       int returnCode=relay_send_command_from_edge(CONTROL_CELL_ID, TO_CIRCUIT(circ),
                                   RELAY_COMMAND_INTRODUCE1,
                                   (const char*) cell_out, length,
↪  circ->cpath->prev);
       if (returnCode) {
-        log_warn(LD_GENERAL, "TH: Unable to send introduce cell on circuit %u",
↪  TO_CIRCUIT(circ)->n_circ_id);
+        log_err(LD_GENERAL, "TH: Unable to send introduce cell on circuit %u",
↪  TO_CIRCUIT(circ)->n_circ_id);
+        control_write_endreply(conn, 400, "Error: Could not send introduce cell on
↪  provided circuit");
+        return -1;
       } else{
         circuit_change_purpose(TO_CIRCUIT(circ),
↪  CIRCUIT_PURPOSE_C_INTRODUCE_ACK_WAIT);
       }
     }
-
-    //int status = hs_client_send_introduce1(circ, &rend_circ_mock);
+
     log_info(LD_GENERAL, "Intro1 cell should have been sent");
     send_control_done(conn);
     return 0;
@@ -2420,7 +2477,7 @@ static const control_cmd_def_t CONTROL_COMMANDS[] =
   ONE_LINE(onion_client_auth_remove, 0),
   ONE_LINE(onion_client_auth_view, 0),
   MULTLINE(posthsdescriptor, 0),
-  ONE_LINE(posthssub, 0),
+  MULTLINE(posthssub, 0),
   ONE_LINE(launch_by_extend, 0),
 };

diff --git a/src/feature/hs/hs_cache.c b/src/feature/hs/hs_cache.c
index cf8e377313..ed8c93c028 100644
--- a/src/feature/hs/hs_cache.c
+++ b/src/feature/hs/hs_cache.c
@@ -412,7 +412,7 @@ remove_v3_desc_as_client(const hs_cache_client_descriptor_t *desc)
 }

 /** Store a given descriptor in our cache. */
-static void
+void
 store_v3_desc_as_client(hs_cache_client_descriptor_t *desc)
 {
   hs_cache_client_descriptor_t *cached_desc;
```

```
diff --git a/src/feature/hs/hs_cache.h b/src/feature/hs/hs_cache.h
index dd55f54ba4..fb6c8130a2 100644
--- a/src/feature/hs/hs_cache.h
+++ b/src/feature/hs/hs_cache.h
@@ -148,6 +148,9 @@ STATIC size_t cache_clean_v3_as_dir(time_t now, time_t
↪  global_cutoff);
 STATIC hs_cache_client_descriptor_t *
 lookup_v3_desc_as_client(const uint8_t *key);

+void
+store_v3_desc_as_client(hs_cache_client_descriptor_t *desc);
+
 #endif /* defined(HS_CACHE_PRIVATE) */

 #endif /* !defined(TOR_HS_CACHE_H) */
diff --git a/src/feature/hs/hs_cell.c b/src/feature/hs/hs_cell.c
index 152aa5fc6c..01a6e43342 100644
--- a/src/feature/hs/hs_cell.c
+++ b/src/feature/hs/hs_cell.c
@@ -407,20 +407,6 @@ introduce1_set_encrypted(trn_cell_introduce1_t *cell,
   //TH: TODO Need to add functionality here to allow setting extensions.
   ext = trn_cell_extension_new();
   tor_assert(ext);
-  //TH: Create 3 fields for condition, callback and descriptor
-  trn_cell_extension_field_t *condition = trn_cell_extension_field_new();
-  trn_cell_extension_field_t *callback = trn_cell_extension_field_new();
-  trn_cell_extension_field_t *descriptor = trn_cell_extension_field_new();
-  //TH: Set values:
-  cell_extension_set_field_to_string(condition, "Face: 123123124123123123", 31);
-  cell_extension_set_field_to_string(callback, "blablablablabla.onion", 32);
-  cell_extension_set_field_to_string(descriptor, "here could be a service
↪  descriptor", 33);
-  //TH: Now set extension size and add to values
-  trn_cell_extension_set_num(ext, 3);
-  trn_cell_extension_setlen_fields(ext, 3);
-  trn_cell_extension_set_fields(ext, 0, condition);
-  trn_cell_extension_set_fields(ext, 1, callback);
-  trn_cell_extension_set_fields(ext, 2, descriptor);
   trn_cell_introduce_encrypted_set_extensions(enc_cell, ext);

   /* Set the rendezvous cookie. */
@@ -443,7 +429,7 @@ introduce1_set_encrypted(trn_cell_introduce1_t *cell,
   trn_cell_introduce_encrypted_free(enc_cell);
 }

-
+static void
 introduce1_set_extended_encrypted(trn_cell_introduce1_t *cell,
                           const hs_cell_introduce1_data_t *data,
                           trn_cell_extension_t *private_extensions)
@@ -929,12 +915,41 @@ hs_cell_parse_introduce2(hs_cell_introduce2_data_t *data,
     }
   }

   /* XXX: Implement client authorization checks. */
   //TH: First draft of my own extension handling.
   trn_cell_extension_t *extensions=
↪  trn_cell_introduce_encrypted_get_extensions(enc_cell);
   uint8_t count=trn_cell_extension_get_num(extensions);
```

```
   if (count > 0){
     char *condition,*callback;
+    int contains_descriptor=0;
+    long duration=0l;
+    hs_subcredential_t subcred;
+    ed25519_public_key_t auth_pubkey;
+    curve25519_public_key_t enc_key;
     for(size_t i=0; i<count; i++) {
       trn_cell_extension_field_t *field = trn_cell_extension_get_fields(extensions,
↪  i);
       uint8_t type=trn_cell_extension_field_get_field_type(field);
@@ -955,11 +970,68 @@ hs_cell_parse_introduce2(hs_cell_introduce2_data_t *data,
           callback=calloc(length+1, sizeof(char));
           strcpy(callback, content);
           break;
-       case 33:break; //TODO: Implement logic to load descriptor if descriptor is
↪  present. Need to verify size restrictions first
+       case 33:
+         duration=atol(content);
+         break;
+       case 34:
+         for (int j=0;j<32;j++){
+           subcred.subcred[j]=(uint8_t)content[j];
+         }
+         contains_descriptor=1;
+         break;
+       case 35:
+         for (int j=0;j<32;j++){
+           auth_pubkey.pubkey[j]=(uint8_t)content[j];
+         }
+         contains_descriptor=1;
+         break;
+       case 36:
+         for (int j=0;j<32;j++){
+           enc_key.public_key[j]=(uint8_t)content[j];
+         }
+         contains_descriptor=1;
+         break;
        default: log_warn(LD_GENERAL, "TH: Detected unknown Extension with Type %i and
↪  %lu bytes of data: %s", type, length, content);
       }
     }
-    //TH: Now send the descriptor event, if possible
+    //TH:Put the included descriptor in the hs_cache, if there is any
+    …
     if (condition && callback){
       log_info(LD_GENERAL, "TH: Received an HSSUB-Extension");
       control_event_hs_sub(condition, callback);
@@ -968,7 +1040,6 @@ hs_cell_parse_introduce2(hs_cell_introduce2_data_t *data,
       goto done;
     }
   }
-

   /* Extract onion key and rendezvous cookie from the cell used for the
    * rendezvous point circuit e2e encryption. */
@@ -983,14 +1054,14 @@ hs_cell_parse_introduce2(hs_cell_introduce2_data_t *data,
   for (size_t idx = 0;
        idx < trn_cell_introduce_encrypted_get_nspec(enc_cell); idx++) {
```

```
     link_specifier_t *lspec =
-      trn_cell_introduce_encrypted_get_nspecs(enc_cell, idx);
+    trn_cell_introduce_encrypted_get_nspecs(enc_cell, idx);
     if (BUG(!lspec)) {
       goto done;
     }
     link_specifier_t *lspec_dup = link_specifier_dup(lspec);
     if (BUG(!lspec_dup)) {
       goto done;
-    }
+      }
     smartlist_add(data->link_specifiers, lspec_dup);
   }

@@ -1093,7 +1164,6 @@ hs_cell_build_extended_introduce1(const
 ↪  hs_cell_introduce1_data_t *data,
 {
   ssize_t cell_len;
   trn_cell_introduce1_t *cell;
-  trn_cell_extension_t *ext;

   tor_assert(data);
   tor_assert(cell_out);
@@ -1113,8 +1183,7 @@ hs_cell_build_extended_introduce1(const
 ↪  hs_cell_introduce1_data_t *data,
   introduce1_set_extended_encrypted(cell, data, private_extensions);

   /* Final encoding. */
-  cell_len = trn_cell_introduce1_encode(cell_out, RELAY_PAYLOAD_SIZE, cell);
-
+  cell_len = trn_cell_introduce1_encode(cell_out, 16384, cell);
   trn_cell_introduce1_free(cell);
   return cell_len;
 }
diff --git a/src/feature/hs/hs_cell.h b/src/feature/hs/hs_cell.h
index 5be2c2e656..ddc5bb0eb4 100644
--- a/src/feature/hs/hs_cell.h
+++ b/src/feature/hs/hs_cell.h
@@ -13,6 +13,10 @@
 #include "feature/hs/hs_service.h"
 #include "trunnel/hs/cell_common.h"

+#define HS_CACHE_PRIVATE
+#include "feature/hs/hs_cache.h"
+#include "src/feature/nodelist/nodelist.h"
+
 /** An INTRODUCE1 cell requires at least this amount of bytes (see section
  * 3.2.2 of the specification). Below this value, the cell must be padded. */
 #define HS_CELL_INTRODUCE1_MIN_SIZE 246
```

Listing B.13 shows the code needed to extract the relevant fields from a valid service descriptor to include a minimal service descriptor within an *INTRODUCE1* cell. There are two ways how a service descriptor can be specified to be used as a callback. If it has already been loaded into the Tor client's cache, it is sufficient to specify the onion address, otherwise the descriptor has to be included as request data, just like it is done when loading service descriptors. Once the descriptor has been found, we select a random introduction point from it, extract the information for a minimal descriptor, and put it in exten-

sion fields that can be forwarded to the onion service.

Listing B.13: Insert callback information control_cmd.c

```
2237 ed25519_public_key_t callback_public_key;
2238     hs_descriptor_t *callback_descriptor=NULL;
2239     if (hs_parse_address(callback->value, &callback_public_key, NULL, NULL) != 0){
2240       log_err(LD_GENERAL, "Could not parse provided callback address. Aborting
     ↪ HSPOST");
2241       control_write_endreply(conn, 400, "Error: Your provided callback address could
     ↪ not be parsed");
2242       return -1;
2243     }
2244     if(cmd_args->cmddata_len > 0) {
2245       log_debug(LD_GENERAL, "We received a callback descriptor, try to pack
     ↪ descriptor in INTRODUCE1");
2246 
2247       //OK, we have a valid onion address as callback
2248       hs_client_decode_descriptor(cmd_args->cmddata, &callback_public_key,
     ↪ &callback_descriptor);
2249       if (callback_descriptor == NULL){
2250         control_write_endreply(conn, 400, "Error: Your provided descriptor data
     ↪ could not be parsed");
2251         return -1;
2252       }
2253     } else{
2254       log_debug(LD_GENERAL, "We received a callback address without descriptor.
     ↪ Checking if descriptor is in cache:");
2255       callback_descriptor=(hs_descriptor_t*)hs_cache_lookup_as_client(
     ↪ &callback_public_key );
2256     }
2257 
2258     if (callback_descriptor) {
2259       hs_desc_intro_point_t* ip=NULL;
2260       ip = (hs_desc_intro_point_t*)
     ↪ smartlist_get(callback_descriptor->encrypted_data.intro_points, 0);
2261 
2262       if(ip == NULL){
2263         control_write_endreply(conn, 400, "Error: You specified a service descriptor
     ↪ without introduction points");
2264         return -1;
2265       }
2266       ed25519_public_key_t sign_key= ip->auth_key_cert->signed_key;
2267       hs_subcredential_t subc = callback_descriptor->subcredential;
2268       intro1_data.link_specifiers=ip->link_specifiers;
2269       intro1_data.client_kp=&rend_pubkey;
2270       //TODO I have 20 bytes of data left I can encode in there
2271       intro1_data.rendezvous_cookie=rend_cookie;
2272 
2273       //Set up extensions that only get added in this case:
2274       trn_cell_extension_set_num(private_extensions, 6);
2275       trn_cell_extension_setlen_fields(private_extensions, 6);
2276       trn_cell_extension_field_t *callback_subcredential =
     ↪ trn_cell_extension_field_new();
2277       trn_cell_extension_field_t *callback_authkey = trn_cell_extension_field_new();
2278       trn_cell_extension_field_t *callback_enc_key = trn_cell_extension_field_new();
2279       cell_extension_set_field_to_bytes(callback_subcredential, 32, subc.subcred,
     ↪ 34);
2280       cell_extension_set_field_to_bytes(callback_authkey, 32, sign_key.pubkey, 35);
```

```
2281      cell_extension_set_field_to_bytes(callback_enc_key, 32,
    ↪  ip->enc_key.public_key, 36);
2282      trn_cell_extension_set_fields(private_extensions, 3, callback_subcredential);
2283      trn_cell_extension_set_fields(private_extensions, 4, callback_authkey);
2284      trn_cell_extension_set_fields(private_extensions, 5, callback_enc_key);
2285
2286    }else{
2287      intro1_data.link_specifiers = intro_point->link_specifiers;
2288      intro1_data.rendezvous_cookie=rend_cookie;
2289      intro1_data.client_kp=&rend_pubkey;
2290
2291      trn_cell_extension_set_num(private_extensions, 3);
2292      trn_cell_extension_setlen_fields(private_extensions, 3);
2293    }
```

The final task that remains is creating a valid service descriptor object from the extension data received by the onion service. Listing B.14 shows the code added to achieve this goal. The extension data includes the required keys and link identifiers and from that data a *hs_descriptor_t* struct is created and inserted into Tor's internal descriptor cache.

Listing B.14: Build minimal descriptor from callback info hs_cell.c

```
997   //TH:Put the included descriptor in the hs_cache, if there is any
998   if(contains_descriptor>0){
999     log_info(LD_GENERAL, "Received HSSUB with callback-descriptor, building
    ↪  descriptor now");
1000    hs_descriptor_t *descriptor=tor_calloc(1, sizeof(hs_descriptor_t));
1001    hs_cache_client_descriptor_t
    ↪  *desc_entry=tor_calloc(1,sizeof(hs_cache_client_descriptor_t));
1002
1003    desc_entry->encoded_desc="MINIMAL_DESCRIPTOR";
1004    desc_entry->expiration_ts=(time_t) duration;
1005    hs_parse_address(callback, &desc_entry->key, NULL, NULL);
1006    descriptor->plaintext_data.lifetime_sec=3600;
1007    descriptor->subcredential=subcred;
1008    hs_desc_intro_point_t *ip = hs_desc_intro_point_new();
1009    ip->auth_key_cert = tor_calloc(1, sizeof(tor_cert_t));
1010    ip->auth_key_cert->signed_key=auth_pubkey;
1011    smartlist_t *ip_link_specifiers=smartlist_new();
1012    ed25519_public_key_t node_id;
1013    SMARTLIST_FOREACH_BEGIN(data->link_specifiers,
1014                            const link_specifier_t *, ls) {
1015      link_specifier_t* ls_intro=link_specifier_dup(ls);
1016      if (link_specifier_get_ls_type(ls_intro) == 3) {
1017          memcpy(node_id.pubkey,
1018          link_specifier_getconstarray_un_ed25519_id(ls_intro),
1019          ED25519_PUBKEY_LEN);
1020      }
1021      smartlist_add(ip_link_specifiers, ls_intro);
1022    } SMARTLIST_FOREACH_END(ls);
1023    ip->link_specifiers=ip_link_specifiers;
1024    ip->enc_key=enc_key;
1025    const node_t *ip_node = node_get_by_ed25519_id(&node_id);
1026    ip->onion_key= *node_get_curve25519_onion_key(ip_node);
1027    smartlist_t *intro_points=smartlist_new();
```

```
1028      smartlist_add(intro_points, ip);
1029      descriptor->encrypted_data.intro_points=intro_points;
1030      desc_entry->desc=descriptor;
1031      store_v3_desc_as_client(desc_entry);
1032    }
1033    log_info(LD_GENERAL, "Descriptor was stored successfully");
1034    //TH: Now fire the descriptor event, if possible
1035    if (condition && callback){
1036      log_info(LD_GENERAL, "TH: Received an HSSUB-Extension");
1037      control_event_hs_sub(condition, callback);
1038      free(condition);
1039      free(callback);
1040      goto done;
1041    }
1042  }
```