Technische Universität Berlin

EINSTEIN CENTER Digital Future

Denial of Service Defenses for Onion Services

Thesis zur Erlangung des Grades
Master of Science (M. Sc.)
im Studiengang Computer Science

Valentin Franck
valentin.franck@campus.tu-berlin.de

Distributed Security Infrastructures
Institut für Softwaretechnik und Theoretische Informatik
Fakultät Elektrotechnik und Informatik
Technische Universität Berlin

Gutachter:
Prof. Dr. Florian Tschorsch
Prof. Dr. Jean-Pierre Seifert

eingereicht am: 12. März 2020

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, 12. März 2020

.....

(Valentin Franck)

Zusammenfassung

Tor Onion Services sind anfällig für Denial-of-Service-Angriffe. Diese Angriffe funktionieren, weil Angreifer relativ wenig Ressourcen benötigen, um einen Onion Service mit Introduce Cells zu überschwemmen, was dazu führt, dass der Dienst aufwändige Rechenoperationen durchführt, um die Rendezvous Circuits zu konstruieren. Dadurch wird die CPU des Dienstes überlastet. In dieser Masterarbeit stellen wir eine Erweiterung der Tor-Rendezvous-Spezifikation vor, die Onion Services vor dem Angriff schützt. Dazu konfiguriert der Onion Service Grenzwerte für Introduce Cells, so dass Introduce Cells ohne Zugangstoken vom Onion Service verworfen werden, sobald die Grenzwerte erreicht werden. Die benötigten Tokens, um sich trotz der Grenzwerte zu verbinden, erhalten legitime Clients vom Onion Service, nachdem sie eine ausreichend schwierige Challenge gelöst haben. Die Tokens sind nicht miteinander oder mit dem Client verknüpfbar. Dieser Ansatz verhindert die Überlastung der CPU des Onion Services und schützt die Verfügbarkeit für legitime Clients. Kryptographisch basiert das Protokoll auf verifizierbaren Oblivious Pseudo Random Functions (V-OPRFs), die auf einer elliptischen Kurve berechnet werden. Unsere Benchmarks zeigen, dass das Protokoll sowohl beim Signieren als auch bei der Überprüfung signierter Tokens effizient arbeitet. Darüber hinaus zeigt die Auswertung, dass ein Angreifer – je nach verwendeter Hash-Funktion – mit unserer Protokollerweiterung bis zu 62,4 oder 47,8 Mal mehr Introduce Cells als ohne die Erweiterung senden muss, um einen erfolgreichen Denial-of-Service-Angriff durchzuführen.

Abstract

Tor onion services are vulnerable to Denial of Service attacks. These attacks work because attackers need relatively few resources to flood an onion service with introduce cells causing the service to perform expensive operations in order to construct the rendezvous circuits. This eventually exhausts the service's CPU. In this master thesis, we present an extension to the Tor Rendezvous Specification that mitigates the attack. The onion service configures rate limits for introduce cells, so that introduce cells without tokens are discarded by the onion service, if they exceed the rate limits. Legitimate clients can retrieve tokens from the onion service after solving a sufficiently hard challenge. These unlinkable tokens allow the client to connect to the onion service even if the rate limits are exceeded. This approach mitigates the CPU time exhaustion at the service and protects the service's availability for legitimate clients. Cryptographically, the protocol is based on verifiable oblivious pseudo random functions (V-OPRFs) computed over an elliptic curve. Our benchmarks show that the protocol performs efficiently for both token issuance and redemption. Furthermore, the evaluation shows that, depending on which hash function is used, with our defenses an attacker has to send up to 62.4 or 47.8 times more introduce cells to mount a successful Denial of Service attack than without our defenses.

Contents

Chapter 1

Introduction

Onion services allow operators to provide their services, for example a website, anonymously within the Tor network. Currently, onion services are vulnerable to Denial of Service (DoS) attacks. These attacks work because an attacker needs relatively few resources to flood an onion service with introductions causing the service to perform expensive operations in order to construct the rendezvous circuits. This eventually exhausts the service's CPU. In this thesis, we present an extension to the Tor Rendezvous Specification that mitigates the attack. In the remainder of the introduction chapter we explain our motivation, our contribution in more detail and how the rest of this thesis is organized.

1.1 Motivation

Anonymous communication is one of the bases of a democratic society in the digital age. It is vital for the protection of fundamental civil rights such as the right to freedom of speech and the right to privacy.[1] Recent revelations have shown how hard it is to achieve anonymity, given that both states and private actors seek to monitor and analyze user behavior online. In recent years, states tend to restrict citizen rights in the fight against terrorism and mount large surveillance programs, while private companies are more interested in the collection of user data for the sake of selling targeted advertisements [Art15; Wes19]. Therefore, surveillance represents a serious threat for actors such as activists and journalists (e.g. in repressive regimes) that must rely on their anonymity or the anonymity of their sources. This is why Tor has become a crucial tool in the defense of democracy and human rights. The objective of our work is to make Tor more resilient and secure to use for anybody in need of anonymous communication.[2]

Currently, Tor is the most widely used anonymity network with more than two million users at any time [Tor20]. Tor is a low latency anonymity network which means it allows users to browse the web or run live protocols such as SSH, Telnet, IRC etc. This distinguishes it from other anonymity networks such as mix networks, that provide stronger anonymity at the price of a higher latency. Compared to mix networks Tor has a different threat model. It does not protect anonymity against a global passive adversary [DMS04].

Tor does not only allow client users to access services on the Internet anonymously, it also enables users to host their services protecting both the client's and the server's anonymity and making both end points unlinkable. These services are called onion services and can only be accessed from within the Tor network. Originally, onion services were named hidden services (see [DMS04]). The name has

[1] For an elaborate argument on why online anonymity should be a fundamental right see [Art15].

[2] Of course, the network has also been used for criminal activities. Nevertheless, we consider anonymity networks legitimate and anonymity a fundamental right.

been changed and the underlying protocol is subject of ongoing development. In this thesis, we refer to version 3 of the Tor Rendezvous Specification, which defines how onion services are provisioned [Tor19].[3]

1.2 Contribution

We present a new extension to the Tor Rendezvous Specification, that allows onion services to mitigate CPU based DoS attacks and remain available even when such attacks happen. With the introduction of a token-based access system onion services are able to distinguish between legitimate and malicious traffic. This new system is made of well-known cryptographic building blocks. We modify the cryptographic scheme used in Privacy Pass [Dav+18] and apply it to the context of onion services by making the required modifications to both cryptography and the Rendezvous Specification. Our design has several advantages over previous solutions aiming to mitigate DoS attacks. Most noteworthy, it protects the availability of onion services targeted by DoS attacks instead of only protecting the health of the Tor network in general.

Our approach is based on a non-interactive zero-knowledge proof (ZKP), that is used to generate access tokens for legitimate clients. Cryptographically, we use a verifiable oblivious pseudo random function (V-OPRF) in order to issue and redeem unlinkable blinded tokens. Initially, the onion service configures rate limits for `INTRODUCE2` cells (see Section 2.2), so that `INTRODUCE2` cells without a valid access token are discarded by the onion service, as soon as the rate limits are exceeded. In short—omitting some parts needed for the security of the protocol—token issuance and redemption work as follows. The client connects to the onion service (while the service is not under attack) and retrieves a challenge from the onion service. The client solves the challenge and prepares blinded tokens. The client sends the solution and the blinded tokens to the onion service. The service verifies the solution and signs the tokens with a secret key, i.e. it the service computes the OPRF output. The blinded signed tokens are sent back to the client. The client unblinds the tokens and stores the signatures and the tokens for future use. The next time the client connects to the onion service it adds a signed token to the `INTRODUCE2` cell. The service validates the signature on the token and establishes the connection. Any `INTRODUCE2` cell without a valid token will be rejected by the onion service, when it is under attack, i.e. the rate limits for `INTRODUCE2` cells without tokens are exceeded.

Because the protocol is based on elliptic curve cryptography it is efficient compared to asymmetric schemes such as RSA. Furthermore, we present SMG, a new function to hash numbers to points on the elliptic curve NIST P-256, which we use in our implementation for OPRF computation. SMG reduces the run times for the two operations that use it in our cryptographic scheme to 70% compared to Simplified SWU, the hash function which is used in Privacy Pass for these operations. This represents a major improvement of the performance of the protocol and in particular of the critical token validation operation which is run by the onion service.

Our DoS defenses protocol does not require any additional round trip times (RTTs) for token redemption. The required data can simply be added to the cells that are exchanged by the protocol. Only two RTTs are required for token issuance, one RTT for sending and resolving the challenge and one RTT for the signing of the actual tokens. Token issuance happens simultaneous to regular application layer traffic. This ensures that our approach does not introduce additional latency. Because tokens are small in size, only minimal to no additional bandwidth is required.

Tor's security properties are not weakened, i.e. none of the involved parties learns any new information about each other. Tokens requested in a single batch are not linkable to one another, once they are unblinded and neither can a token redemption be linked to its request, i.e. tokens are not linkable to

[3] Theoretically, it should be possible to adopt our design for version 2 onion services. Nevertheless, considering the improved security properties of version 3 we decided against discussing the implications of our design for version 2 of the Tor Rendezvous Specification.

a specific user identity. No (semi-)trusted third party or user registration processes are required. The onion service operator has full control over the degree of DoS attack protection it gets, by adjusting the according parameters and the hardness of the challenge. No additional trust in any other part of the Tor network is required by neither the client nor the onion service. The changes to the Rendezvous Specification remain minimal and are backward compatible.

In contrast to DoS mitigation based on rate limits at the introduction point (IP) legitimate traffic is never throttled. Thus, we preserve a high level of usability and service availability for legitimate users. This does not only protect the service itself but as a side effect also the health of the Tor network in general. Up to this point, protecting service's availability has been an unresolved problem.

1.3 Organization

The remainder of this thesis is organized as follows. The next chapter provides background information on Tor and the Rendezvous Specification in particular. This chapter also includes a more detailed description of the problem our thesis resolves, i.e. how CPU based DoS attacks against onion services work and why they are such a severe threat to onion service availability. In Chapter 3 we discuss related work. This chapter includes sections on previous research on DoS attacks against Tor and potential DoS detection systems. It also includes a section on the cryptography needed for our DoS defenses. We decided to place the chapter on related work at this location, because the information given in the background chapter is vital to understand some of the work discussed here. Chapter 4 presents our threat model and the design (and security) goals defined for our DoS defenses. Before presenting the details of the design of our DoS defenses in Chapter 6, we discuss three potential ways to mitigate DoS attacks against onion services. We placed this discussion prior to the presentation of our own approach because it helps making our design decisions clear and it points out which crucial advantages our DoS defenses have over other ideas. Chapter 7 contains the details of the implementation of our prototype. The chapter describes, how the the approach was integrated into the existing Tor implementation, including changes to the Rendezvous Specification and definition of new cell formats. In Chapter 8 we evaluate our DoS defenses design. This includes an informal discussion of the security properties defined in Chapter 4 and performance benchmarks for all operations required by our design. We finish the thesis with a conclusion.

Chapter 2

Background

In this chapter we will describe the background of our thesis. First we provide a brief summary of how the Tor network works and what its main components are. Next we will summarize the relevant details of the Tor Rendezvous Specification, as it contains the most relevant protocols for our thesis and describes how clients and onion services can talk to each other (anonymously). In the final section we provide a description of the problem which this thesis solves. In this section we explain how Denial of Service (DoS) attacks based on CPU exhaustion of the targeted onion service work. This is the class of attack the DoS defenses presented in Chapter 6 are designed to defend against. We decided to place this chapter before the chapter on related work, because the research discussed in *Related Work* is more understandable, if one is familiar with the background of Tor and the Rendezvous Specification.

2.1 Tor Network

Tor is a low-latency anonymity network, that uses onion routing to disguise the user's identity and location for TCP-based applications.[1] The Tor design was presented in [DMS04]. The explanations in this section are based on this paper unless other references are provided. Onion routing is an overlay network on top of the Internet. Tor aims at finding a reasonable tradeoff between anonymity and usability, i.e. there are more secure (high-latency) anonymity networks, but they are not usable for the same type of applications such as modern web browsing. Tor has the largest user base of all anonymity networks, meaning that Tor users are part of a relatively large anonymity set of at least 2 million daily users [Tor20]. Apart from anonymization of communication, Tor is also used to circumvent censorship. We will not present the respective subsystems here, because they are not relevant for this thesis. Also, we will not present the applications closely related to the Tor network such as the Tor Browser Bundle, whose goal it is to provide secure anonymous web browsing.

Onion routing: In onion routing the client selects a path of nodes over which traffic is routed before it reaches the destination. The client adds multiple layers of encryption. Each node on the selected path removes one layer of encryption before forwarding the packet to the next node. Hence, the packet is "peeled" like an onion. When the response is sent back to the client, the opposite happens: each node on the path adds one layer of encryption. Only the client has the keys to remove all layers of encryption and read the data in plaintext. Nodes can only see their predecessor and their successor, but never the complete path. Only the exit node can see the data in plaintext, unless of course, the application layer uses encryption (e.g. HTTP over TLS).

Tor relays: The servers over which traffic is routed in the Tor network are called onion routers or

[1] Running applications that are not TCP-only over Tor may leak a user's IP address. Tor supports domain name resolution over TCP.

relays. Currently, the network consists of about 6500 relays [Tor20]. Relays are very diverse in terms of the bandwidth they contribute. Clients retrieve the signed list of all relays from the directory servers. There are ten directory servers, that are trusted. The list of relays contains flags and weights in order to ensure that clients choose relays as part of a path according to their capacities. The most reliable relays are flagged as guard nodes. These are used by clients as entry nodes and remain the same over a longer period of time, because this minimizes the probability of selecting a malicious guard, which would make deanonymization much easier. Only a part of the relays (1100) can be used as exit nodes [Tor20]. Depending on legislation exit nodes may be accountable for the traffic that leaves the Tor network through them, which is why relay operators carefully decide whether they want to run an exit node. Exit nodes can see the destination of the traffic and if the application layer does not encrypt traffic it is in plaintext.

Clients: Clients (usually) do not forward traffic in the Tor network and therefore are no relays. Clients run an onion proxy to connect to relays and route traffic through the Tor network (and receive the respective responses). In order to anonymize communication the user configures the application to use the SOCKS proxy of the Tor process for all network traffic. Tor only supports TCP-based applications.

Note, technically, onion services are also Tor clients, because they use an onion proxy to connect to the Tor network. Nevertheless, in this thesis we distinguish onion services and clients. On the application layer above Tor, onion services act as servers (e.g. web servers). Therefore, by clients we only refer to those Tor instances that behave as clients on the application layer (e.g. web browsers) and use Tor to connect to a server.

Circuits: Tor is based on circuits. Usually circuits consist of three hops: an entry node, a middle node and an exit node. When a client connects to a server, it first selects a path over several relays and builds the circuit step by step. All connections between relays, and between relays and clients are TLS-encrypted. Once the circuit is setup, multiple TCP streams can be sent through it. This means that Tor has an initial delay because circuits first have to be constructed before any traffic can go through the network. Once the circuit is constructed, all traffic takes the same route in both directions. (Of course, clients can construct more than one circuit and use different circuits in order to avoid traffic correlation attacks.)

Cells: In Tor all traffic is passed as cells of a fixed size: 512 bytes. Cells consist of a header and payload. There are two types of cells: control cells and relay cells. Control cells are handled by the node that receives them. Their header contains the circuit ID (2 bytes) and the control command (1 byte). For example a `CREATE` cell is used to setup a new circuit. Relay cells carry end-to-end data. They have several additional header fields, all of which are encrypted together with the payload: stream ID (2 bytes), a digest for integrity checking (6 bytes), the length of the payload (2 bytes) and the relay command (1 byte). Relay cells are used for signaling and to send data down the stream (`RELAY_DATA` cells). An example of cells used for signaling are `RELAY_EXTEND` cells. The client sends these cells over an existing circuit to extend it (usually during circuit construction). The last hop in the circuit (that removes the last layer of encryption) will interpret the cell and send a `CREATE` cell to the next hop, which was indicated in the payload of the `RELAY_EXTEND` cell. We will describe the relay commands of the Rendezvous Specification in the next section.

2.2 Tor Rendezvous Specification

In this section we will describe what onion services are and how they can be accessed. Essentially, an onion service is a regular service, that can be hosted on a server (like a web server or an email server) with the difference that it runs as a Tor client and can only be accessed from within the Tor network, thus allowing both client and onion service to remain anonymous. Onion services have been specified in [Tor19]. Figure 2.1 gives an overview over the protocol. In more detail the steps of the Rendezvous

Specification are the following.[2] Before describing the individual steps of the rendezvous protocol we explain what hidden service directories (HSDirs) are. They were not explained in the previous section, because they are only relevant for the Rendezvous Specification.

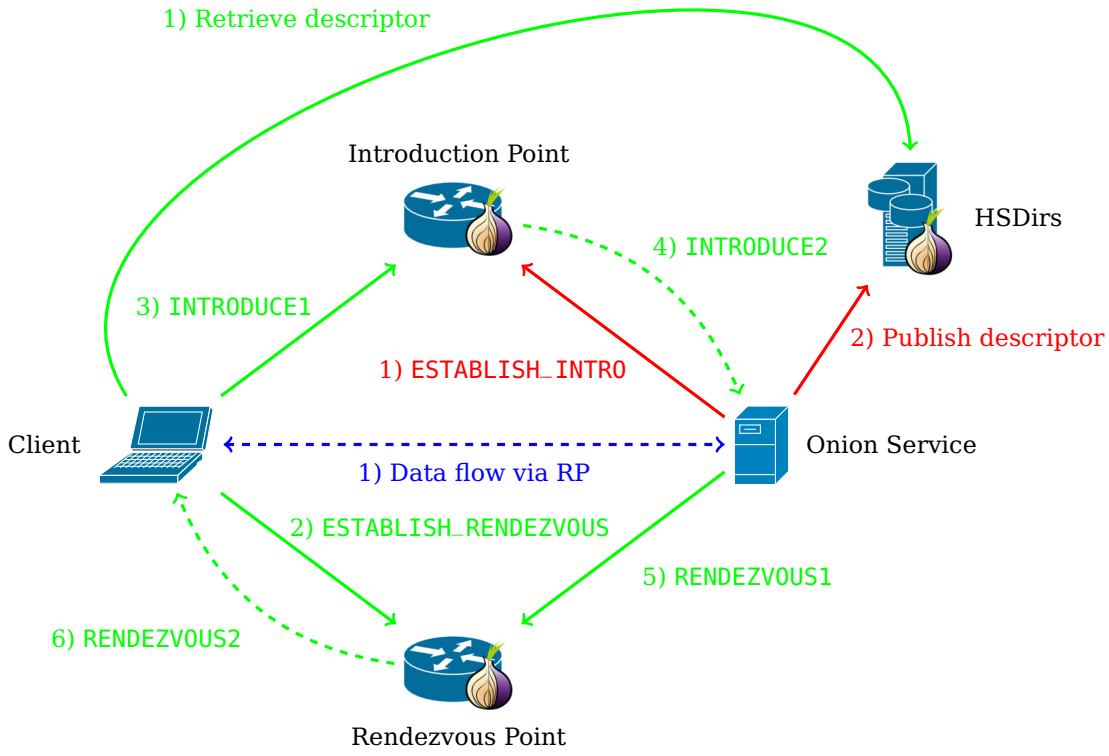

Figure 2.1: A client connects to an onion service. All arrows represent traffic, that goes through Tor circuits. The direction of the arrow (from source to destination) indicates who constructs the circuit. Dashed lines indicate, that cells are sent over a preexisting circuit. The setup phase of the onion service is displayed in red. The connection establishment initiated by the client is displayed in green. The data flow between client and onion service via the RP, after the connection is established, is displayed in blue.

Hidden service directories (HSDirs): Onion services compose the information clients need to connect to them in the onion service descriptors and periodically upload these descriptors to a set of HSDirs. Together, the HSDirs can be understood as a distributed database, that is organized as a hash ring. The position of each HSDir is based on a random value and the HSDir's public identity keys. This random value is agreed upon in a voting of the directory servers and is part of the network consensus. It changes during each period. Therefore, the position of each HSDir in the hash ring changes periodically. Onion services choose the positions to place their descriptors into the hash ring based on the keys they used to sign the descriptors. These keys also change during each period. Clients can derive which key was used for signing, if and only if they know the onion address (which by itself is based on the onion service's long term public identity key). The changing topology of the hash ring

[2] This is a simplified version of the protocol. For example the details of key derivation and descriptor based access controls are given in [Tor19]. Since those parts of the protocol are not of immediate relevance for this thesis, we do not describe them here.

and the fact that onion service descriptors are uploaded in a redundant manner protects this part of the protocol from DoS attacks [Tor19].

Other important entities in the Rendezvous Specification are introduction points (IPs) and rendezvous points (RPs). IPs allow clients to introduce themselves to the onion service and establish a connection via an RP, so that both sides may remain anonymous. The exact functions of IPs and RPs becomes clear from the following overview over the Rendezvous Specification:

1. The onion service picks relays from the network, called introduction points (IPs) and builds circuits to them, telling the IPs its public key. The cells it sends to the IPs are called `RELAY_COMMAND_ESTABLISH_INTRO` cells. The IP replies by sending the status of the establishment of the IP in a `RELAY_COMMAND_INTRO_ESTABLISHED` cell.

2. The onion service creates an onion service descriptor signed with the service's private key. Among other data the descriptor contains its public key and the chosen IPs. The descriptor is published in the hidden service directories (HSDirs) as explained above.

3. A client willing to connect to the service downloads the service descriptor. To find the descriptor in the HSDirs the client needs the service's onion address, which is a Base32 encoding of the public key. Because the client can check whether the onion address and the public key retrieved from the descriptor match, the public key is self-authenticating and no further key verification is required to prevent man-in-the-middle attacks.

4. The client establishes two circuits, one to a randomly chosen IP learned from the descriptor and another one to a randomly picked relay, called rendezvous point (RP). The client tells the RP a one time secret in a `RELAY_COMMAND_ESTABLISH_RENDEZVOUS` cell. The RP replies by sending the status of the attempt to establish an RP in a `RELAY_COMMAND_RENDEZVOUS_ESTABLISHED` cell.

5. The client assembles a cell containing the chosen RP, the one time secret and the first part of a cryptographic handshake, encrypts the cell with the onion service's public key and sends it to the IP (`RELAY_COMMAND_INTRODUCE1` cell) asking it to forward the encrypted payload of the cell to the onion service in a `RELAY_COMMAND_INTRODUCE2` cell. The IP acknowledges the receipt of the introduction request by sending a `RELAY_COMMAND_INTRODUCE_ACK` cell back to the client and forwards the cell it received to the onion service as a `RELAY_COMMAND_INTRODUCE2` cell.

6. The onion service decrypts the `RELAY_COMMAND_INTRODUCE2` cell, establishes a circuit to the specified RP and tells it the one time secret in a `RELAY_COMMAND_RENDEZVOUS1` cell. In this cell the onion service also completes the cryptographic handshake with the client, so that the client and the onion service can exchange end-to-end encrypted and authenticated messages via the RP.

7. The RP informs the client of the connection establishment, by forwarding the cell it received from the onion service to the client as a `RELAY_COMMAND_RENDEZVOUS2` cell.

8. Now the client uses `RELAY_BEGIN` cells to establish streams to the onion service. It uses `RELAY_DATA` cells to send data on those streams.

For the sake of readability in the remainder of this thesis we will will drop the leading "`RELAY_COMMAND_`" from the names of the cells. This means we write "`INTRODUCE1` cell" instead of "`RELAY_COMMAND_INTRODUCE1` cell" and so on.

2.3 Problem Statement

In this section we explain the problem solved by this thesis in more detail. The problem, that we refer to, is the protection of the availability of onion services targeted by Denial of Service (DoS) attacks, that

work by exhausting the service's CPU. In the next chapter we will complement the problem statement by giving an overview over previous research on DoS attacks in the Tor network in general.

Due to the large amount of CPU time spent in rendezvous circuit constructions, the onion service protocol is prone to DoS attacks. Those attacks affect both the network in general and the specific onion service targeted by the attack. DoS attacks represent a real threat to the entire network, in particular because they are relatively easy to carry out for an attacker in terms of knowledge and resources that are required.

In early 2018 a severe DoS attack hit the Tor network and in response a DoS mitigation subsystem was designed and implemented. However, this subsystem is not able to protect against all kinds of DoS attacks, in particular those involving onion services [Kad19]. As stated in the next chapter the mitigation of DoS attacks against onion services is still a topic of active research. The simplest way an adversary could carry out a DoS attack against an onion service works as follows.

The attacker creates as many circuits to IPs as possible and tells the onion service to rendezvous at certain relays by sending a lot of `INTRODUCE1` cells to the IPs. Due to the anonymity of the clients that construct circuits to an IP there is no trivial solution to limiting the number of introduction circuits an attacker can create. The IPs will send `INTRODUCE2` cells to the victim onion service, which will extract the information and react by constructing circuits to the specified RPs. Note that the attacker never actually has to create a circuit to the RPs it specifies in its `INTRODUCE1` cells sent to the IPs, because the onion service will only learn that an RP is invalid after constructing the circuit. Hence, the number of circuits created by the attacker (to the IPs) is amplified by the onion service which creates the rendezvous circuits. The selected RP does not even have to be part of the network consensus (propagated by the directory servers), because clients are free to choose a private relay as RP.

The attack causes both heavy load on the network, because each circuit involves three relays, and heavy load on the targeted onion service itself. The attack is particularly effective because the construction of a new circuit requires a great number of expensive operations (in terms of CPU time consumption) of public key cryptography from the onion service. Since the onion service cannot distinguish legitimate and malicious requests, after some point it will not longer be able to provide its service due to CPU exhaustion. Also, the load on the guard relays provoked by the attack might cause the guard relays to go down, exposing the onion service to deanonymization attacks in case it switches to a new set of guard relays.[3] Recently, Tor introduced rate limits at the IPs which help to protect the network against DoS attack, but are useless to protect the availability of onion services targeted by a DoS attacker. Therefore, our DoS defenses focus on the availability of onion services and not the network. Our goal is to avoid the exhaustion of CPU time due to rendezvous circuit construction while at the same time maintaining the service available for legitimate users. Currently, this is not possible, because attackers may easily send enough `INTRODUCE2` cells to exhaust the service's CPU, if no rate limits are set at the IPs. If rate limits are set at the IPs, these will cause the IPs to drop `INTRODUCE1` cells, once the rate limits are exceeded. Throttling traffic indiscriminately at the IPs also means that it becomes unlikely for a legitimate client to establish a connection to the onion service successfully. Therefore, our solution will be the first one that allows legitimate clients to connect to an onion service even while a DoS attack is targeting the service.

[3] For a comprehensive description of such an attack see [ØS06].

Chapter 3

Related Work

In this chapter we will discuss the related work. The first section provides an overview over previous research on Denial of Service (DoS) attacks against the Tor network in general. The research analyzes how these attacks work and how they can potentially be mitigated. A detailed description and current research on the specifics of DoS attacks against onion services was already given in the previous chapter (and will not be repeated here). In the second section of this chapter we discuss research on how to detect and mitigate DoS attacks in Tor. Given that this requires cryptography we include a third section on cryptographic schemes. These cryptographic schemes are an essential building block to defend against DoS attacks in an anonymity network.

3.1 Denial of Service Attacks against Tor

Until a few years ago the problem of DoS attacks against the Tor network or specific targets within the network had been (mostly) neglected by the community. This has changed due to the general increase in DoS attacks on the (public) Internet and also due to the fact that DoS attacks against the Tor network, in the worst case, lead to deanonymization, as they may increase the probability that a victim chooses a compromised relay in her circuits (cf. [Jan+14]).

[Bor+07] provides an analysis of the impacts selective DoS attacks can have on anonymity networks, including the Tor network, under the assumption that the attacker controls enough relays. The study shows that DoS attacks are not only suitable to deny the availability of targeted services but have to be considered a severe threat to anonymity as well. The compromise of anonymity works by denying service to uncompromised circuits, in order to force the victim to use attacker controlled circuits. Tor is able to defend against such attacks with high probability due to the selection of (a set of) guard nodes. [Dan+12] and [DKL09] measure the effectiveness of DoS attacks in Tor and propose two potential detection algorithms. Other than the cited papers, we focus on attacks targeting onion services and not client users that use the Tor network to access the clear net anonymously. Nevertheless, these studies provide valuable insights into the damages DoS attacks can do.

In 2015 Nick Mathewson, one of the original Tor designers, published a paper, that gives an overview over different kinds of DoS attacks against Tor. The paper also discusses possible mitigations for these attacks [Mat15]. In contrast, our goal is not to give a broad overview and general recommendations for future development, but focus on one specific attack and present a practical mitigation for it. This thesis follows one of Mathewson's recommendations, which is to modify the existing protocol such that a DoS attacker cannot spend a small amount of CPU in order to force the onion service to use a lot of CPU [Mat15]. More precisely, our approach focuses on the second part of that finding by protecting onion services from spending CPU time due to DoS attacks.

In 2018 the cited paper by Nick Mathewson was used as the basis for a tech report [Boc+19]. The authors of this tech report for the first time provided a detailed description of DoS attacks against onion services, including an analysis of the first large DoS attack against onion services in December 2017. This DoS attack rendered some onion services and parts of the Tor network unusable. In the case of this attack the authors were able to mitigate the attack at the "guard layer" by limiting a) the rate at which new circuits could be created and b) the number of concurrent connections to a guard relay. Our approach in contrast goes into a different direction, because our primary goal is not DoS attack mitigation at the network level, but protecting onion service availability. Therefore, our approach focuses on changes to the introduction phase of the rendezvous protocol.

In April 2019 the topic was picked up again by George Kadianakis in a message on the Tor developers mailing list [Kad19]. This message initiated a detailed discussion on the mailing list, that inspired the proposed thesis. This discussion, that has been continued up to the submission of this thesis proves the increasing relevance of DoS attack defenses for onion services. As a first consequence the discussion produced Proposal 305 [Gou19], a DoS extension proposal, which introduces new rate limiting parameters at the IP (configured by the onion service in the `ESTABLISH_INTRO` cell) in order to protect the network from illegitimate load due to rendezvous circuit creation. This proposal has been implemented in vanilla Tor in summer 2019. Unfortunately, the approach is not able to protect the availability of a specific onion service targeted by an attack, since legitimate requests are also dropped at the IP due to the rate limits. Thus, legitimate `INTRODUCE1/2` cells have a low probability of reaching the onion service. In contrast, our approach seeks to protect the availability of the onion service itself and only considers protection of the network a welcome side effect, rather than a primary goal.

3.2 Detection of Denial of Service Attacks in Tor

Since Tor provides anonymity it is (also) abused for attacks for example by botnets. This has caused many services to blacklist Tor exit relays, which sometimes makes the network hard to use for legitimate users as they frequently have to solve CAPTCHAs or are blocked entirely. Previous research has been done to identify malicious traffic in Tor. [HG11] surveys a number of different anonymous blacklisting systems and provides a formal definition of anonymous blacklisting systems and which security goals they should achieve. Most of those system are based on some kind of long term user identity, whose introduction to the Tor network, we want to avoid for the sake of anonymity and usability. The security goals for our DoS defenses, defined in Section 4.2, are derived from the goals defined in [HG11].

One example of a DoS detection system is TorPolice [Liu+17], an access control framework enabling providers to throttle malicious traffic without completely shutting down their service for Tor users. Similarly to our design, TorPolice uses blind signatures to preserve the users' privacy. The design introduces a new kind of partially trusted third party, the so called access authorities. In contrast to TorPolice our goal is not to identify malicious traffic leaving the Tor network, but to identify attempts of DoS attacks against onion services, which are part of the network. We also want to avoid the introduction of new partially trusted third parties, because the design of TorPolice allows new kinds of attacks in case the trusted third parties collude. This would be a violation of our security goals, as defined in Section 4.2. Besides these differences, the idea of obtaining a security token using blind signatures in order to get access to a service or network is similar to our design.

While anonymous blacklisting systems try to achieve similar goals by using cryptographic schemes such as blind signatures and OPRFs, the scenario in which they are useful is different from ours. Blacklisting is only useful, if users have to register for a service, that is not free for everyone to be accessed anonymously. Therefore, one might argue that in fact these blacklisting systems actually are whitelisting systems (of registered users), in which it is possible to blacklist users from the whitelist. For the registration a user needs a unique identifier like a scarce resource she spends. Examples are

IP addresses, CAPTCHAs, proofs of work or currency. If such a resource was not required the user could simply re-register to circumvent the blacklisting. This class of attack, where attackers are able to create multiple identities, is called Sybil attack. In our scenario on the other hand, any (legitimate) user should be able to use the onion service without the need for an initial registration. At least, we envision the service to work that way while it is not under attack. This means our design is not based on blacklisting users.

Faust [LH11] is a DoS defense system that changes these blacklisting approaches to provide anonymous whitelisting without the need for a trusted third party. In order to achieve this, Faust uses blind signatures and unlinkable serial transactions [SSG99] with a method to anonymously retrieve the next token. Even though users can remain completely anonymous, Faust requires users to perform some registration, which we want to avoid. Also, our use case differs from Faust, because Faust aims to ban users after they behaved badly by not whitelisting them anymore. This approach is not suitable to mitigate DoS attacks against onion services by users that have not previously registered as in our scenario.

A potential solution to grant only legitimate anonymous users access to a service has been presented in [Dav17] and [Dav+18]: Privacy Pass is designed to enable legitimate users to bypass internet challenges anonymously. This is useful if CDNs (and Cloudflare in particular) block Tor entirely or show CAPTCHAs to Tor users. Privacy Pass uses a verifiable OPRF to enable users to retrieve multiple cryptographic tokens after solving a CAPTCHA. These tokens can be used in future communication with the server so users do not have to solve CAPTCHAs every time but only often enough to discourage DoS attacks. On the client side Privacy Pass is supported by a browser extension. The client has to solve a (CAPTCHA) challenge received from the Cloudflare edge server. The client's solution to the CAPTCHA is then checked by a validator and if correct the client's request is forwarded to the actual web service in order to start the communication. If the client has already solved the CAPTCHA previously it can simply send a token along with its request in order to bypass the CAPTCHA. The tokens which the client receives from the server are cryptographically unlinkable to one another. Also, the signing and the redemption of a token are unlinkable. Additionally, a discrete logarithm equivalence (DLEQ) proof ensures that the same OPRF key is used for every client, so that the server cannot tag individual clients by using different OPRF keys. Privacy Pass performs efficiently and does not produce unacceptable overhead, because it is a 1-RTT protocol. The redemption phase in particular is not expensive in terms of CPU time consumption. Our own solution uses the same cryptographic scheme and is therefore in many aspects similar to Privacy Pass. However, we have to adapt the scheme to make it suitable for DoS protection of onion services. This includes both designing an extension to the Rendezvous Specification and modifications to the OPRF-based cryptographic scheme presented in [Dav+18]. We explain the details of our approach in Chapter 6.

As mentioned before, our design requires some kind of challenge, that clients have to solve in order to obtain access tokens. [Fra+07] presents the Memoryless Puzzle Protocol in order to mitigate Distributed Denial of Service (DDoS) attacks in anonymous routing environments like Tor. The Memoryless Puzzle Protocol pursuits a similar goal, namely reduce the number of decryption operations for onion routers caused by DoS attacks. It also uses a challenge which has to be completed by the client in order to be granted access. However, it integrate the puzzles in the TLS handshake. This is something we want to avoid, because it would disallow the usage of standard cryptographic libraries and it would mean the code had to change for every onion router not just the IP, RP and onion service itself. The solution also causes heavy CPU load on the client side as the puzzles follow a proof of work scheme. This is not acceptable in our opinion, because it will drain the batteries of mobile devices, while attackers with high parallelized computational power might still be able to mount their attacks. Our own solution to this problem abstracts from the specific application layer protocol and lets onion service providers decide for themselves, what kind of challenge they deem adequate. Therefore, in this thesis we will not discuss, which exact challenge should be used. Instead, we will make sure that our protocol works with

all kinds of challenges, so that each onion service operator may use, what she considers appropriate.

A few open questions raised in this section will be answered in the next two chapters. These questions include the following:

1. Can we use the IP in order to sign and redeem blind tokens? How can we do that, if IPs change frequently given that all IPs would need access to the same secret key owned by the onion service?

2. Which cryptographic key should be used for the signing and redemption of access tokens? Can we use the onion service's identity key or a key derived from it?

3. What kind of challenge should clients solve in order to obtain access tokens? How can we make sure there is an appropriate kind of challenge for all onion services? Can we use a CAPTCHA given that onion services do not only provide web-based services, but also protocols such as SSH, SMTP etc. that do not have a GUI? Can we use a proof of work, even for clients on mobile devices?

3.3 Cryptography

If we do not want to base anonymity exclusively on trust as VPN users do, we need cryptography to enforce anonymity technically and make deanonymization infeasible in a given threat model. In the following we discuss the cryptographic schemes, that are the most closely related to our own approach. One such scheme, blind RSA signatures, was invented by David Chaum [Cha83; Cha84]. Blind RSA signatures allow a client to obtain a signature for a message from a server without having the server learn the cleartext message. The client does not learn the server's private key, but after the client unblinds the signature anybody can use the server's corresponding public key to verify the signature of the cleartext message. The crucial new property of blind RSA signatures is that cleartext message and cleartext signature are unlinkable to the blind signature and the blinded message. Many variations of this scheme have been developed to make it applicable for different scenarios and compliant with different security requirements [Gar+11; GG14; FHS15]. In Section 5.2 we discuss in detail how blind RSA signatures could be used in a DoS defense system for onion services, although we finally decided not to use them.

In 2013 a team of Microsoft researchers published a paper presenting a new cryptographic scheme, which they call Keyed-Verification Anonymous Credentials (KVACs) [CMZ14]. We will briefly describe, how this scheme works. The credentials are based on algebraic MACs and zero knowledge proofs (ZKPs). After a setup phase the prover retrieves credentials from the issuer. The prover computes a proof with some randomization and presents it to the verifier. The verifier is now able to tell that the prover received a valid credential but is not able to reveal the identity of the prover. Both issuing and verifying certificates require ownership of the same secret key. Therefore, issuer and verifier are the same entity. In case of Tor onion services this entity would be the onion service. The authors of [CR19] recently published a modified KVACs protocol that allows non-interactive verification. However, we will not further consider KVACs, because these anonymous credentials are based on an underlying long term identity, or at least some identity parameters, the prover has to present to the issuer. Introducing a registration process for clients is undesirable in Tor, given that we want to ensure maximum anonymity. Also, the credentials the prover retrieves are usable for a certain period of time and cannot easily be revoked. It is therefore not a trivial task to modify the protocol, such that the credentials would work in a way that could only allow a limited number of connections to the onion service. Although anonymous authentication is closely related to what we are pursuing and might be a solution for some onion service use cases, we aim for a more general approach, that is not based on an underlying (and unchanged) identity of clients.

Another cryptographic scheme serving a similar purpose like blind RSA signatures are oblivious pseudo random functions (OPRFs). A pseudo random function is a function F, that takes an input x, and

a secret key k in order to generate a pseudorandom output z. We write: $F(k, x) = z$. One possible family of functions F is the HMAC-SHA family. These are *non-oblivious* PRFs. The security goal of oblivious PRFs is that the client receives the output z without learning the key k, while the server must not learn the input x. Recent advances by Jarecki et al. [JKK14; Jar+16] allow the computation of (verifiable) OPRFs in an efficient manner using group based cryptography such as elliptic curve cryptography (ECC). OPRFs based on ECC take a curve point X as input and perform a scalar multiplication with a secret key k. In contrast to HMACs, PRFs using scalar multiplication make it possible to blind the input X by performing a scalar multiplication with a random blinding factor r before sending it to the signer. The multiplicative inverse of the blinding factor r^{-1} can be used to unblind the retrieved output. It is crucial that the signer does not learn the blinding factor, while the secret key must only be known by the signer. The blinding and unblinding thus is similar to blind RSA signatures, with the crucial difference that only one key is used in OPRFs instead of a key pair. The OPRF scheme presented in [JKK14; Jar+16] has been adopted by Privacy Pass, because it allows the client to verify the output of the OPRF (see Section 3.2).

Privacy Pass which is based on verifiable OPRFs uses discrete logarithm equivalence (DLEQ) proofs to provide verifiability of the OPRF to the client. DLEQ proofs were originally presented by Chaum and Pedersen in 1993 [CP93]. They can be understood as a non-interactive zero knowledge proof between a server, the prover, and a client, the verifier. A DLEQ proof prevents a malicious server from tagging users by using more than one key to issue tokens. This is a threat to anonymity as it would allow the server to link different actions of the same user or at least reduce the size of anonymity set by using multiple OPRF keys. DLEQ proofs prevent this by enabling the client to verify that the same key is used for all issued tokens. Modern DLEQ proofs were presented in [JKK14; Jar+16] and adopted by Privacy Pass [Dav+18]. The core idea of a DLEQ proof is to use a zero-knowledge protocol to prove that $log_X(Y) = log_P(Q)$ is true for a pair Y and Q, which has been generated by repeated application of the group operation, i.e. scalar multiplications of the form: $Y = k_1X$ and $Q = k_2P$, where both X and P are generators of the group $\mathbb{G}$ of prime order q. Please note, that in this thesis, we write all operations in their scalar multiplication form in accordance with elliptic curve notation. By proving the equivalence of the discrete logarithm relation the client only learns that the same key has been used to compute Y and Q, i.e. $k_1 = k_2$, but learns nothing about the key itself. It is possible to improve the performance of this scheme by generating the DLEQ proof for a large number of curve points at once [Dav+18]. Because our own scheme is so similar we present a detailed description of the Privacy Pass protocol. For simplification, we will not explain DLEQ proofs here, but refer to Section 6.3 for an exact description of the protocol.

1.
   - In the signing phase the client uses a cryptographic hash function H_1 to compute $X = H_1(x)$, where x is the input. H_1 hashes into a cyclic group $\mathbb{G}$ of prime order q.
   - The client generates a random number r from the ring of integers $\mathbb{Z}_q$.
   - It computes $M = rX$ and sends it to the server.
   - The server computes $Y = kM$, where k is the secret key. It returns Y to the client.
   - The client computes $N = r^{-1}Y = r^{-1}krX = kX$ and stores the pair (x, N) as a token. r^{-1} refers to the multiplicative inverse of r in $\mathbb{Z}_q$.
2.
   - In the redemption phase the client computes request binding data req. This data can be anything related to the specific request by the client such as the requested URL. The server must be able to recompute it as soon as it sees the request.
   - It calculates a shared key $k_s = H_2(x, N)$, where H_2 is another cryptographic hash function, that hashes into a binary string of length n.
   - The client sends $(x, MAC_{k_s}(req))$ to the server, where $MAC_k()$ is a message authentication code using a key k.

- The server also calculates request binding data req' and checks that x has not been used previously before calculating $X' = H_1(x)$, $N' = kX'$ and $k'_s = H_2(x, N')$.
- The server now checks that $MAC_{k'_s}(req')$ equals $MAC_{k_s}(req)$. If the check passes, the token x is considered valid and x is stored so that it cannot be spent again.

This adaptation of the scheme in [JKK14; Jar+16] has first been described in [Dav+18]. In Chapter 6 we explain how this cryptographic scheme allows us to mitigate DoS attacks while preserving the current anonymity and unlinkability properties of the Tor protocol.

One building block of our DoS defenses are challenges that users have to solve in order to obtain tokens and access the onion service during attacks. While we will not analyze what kind of challenges are appropriate in this thesis, here are some considerations regarding the usage of proof of work schemes to solve this problem. The goal of these schemes is to make sure an attacker has to spend enough resources to discourage DoS attacks by adding a challenge for clients. The idea of proof of works has been popularized by Bitcoin [Nak+08], but was originally published by Adam Black in Hashcash [Bac97], whose goal it was to prevent spammers from sending out high numbers of emails without investing many resources. The basic idea behind this scheme is that it is hard to find a hash value with a given property such as being below a certain threshold value, because an attacker has no more efficient option than varying the input until she finds a matching hash value. This requires a lot of computing power. At the same time it is computationally easy to prove a certain input hashes to an accepted value. One problem with this idea is, that for resourceful attackers (e.g. with GPU clusters) it is a lot easier to compute these hashes than for users with regular hardware. For example on mobile devices the computations would quickly drain the battery. Because the calculation of the proof of work consumes so much energy researchers have invented alternatives like proof of stake, proof of activity and proof of space, which are based on different resources but pursue similar goals [SC18]. For us a proof of work is not desirable as a general solution, because it would slow down onion services and make them unusable on mobile devices. Therefore, on mobile devices and for human users it seems to be more acceptable to solve other kinds of challenges such as CAPTCHAs. Nevertheless, a proof of work challenge is interesting for machine based protocols without a user in front of them or for services that offer protocols without a GUI such as SSH. Because it is heavily dependent on the application layer we do not force onion services to use a specific kind of challenge, but rather have the service operators themselves decide what kind of challenge is appropriate. This is possible, because the control port of the Tor process provides the application layer service with an interface to inform the Tor process of correctly solved challenges. For this reason we will not further discuss, which challenge should be used, but assume that there are appropriate challenges for most onion service use cases.

Chapter 4

Design Goals

In this chapter we will present the objective of this thesis. We will first present our threat model, which is derived from the original Tor threat model. We will continue with a detailed description of our security goals. The chapter is finished with a description of our performance goals.

4.1 Threat Model

We adopt the original threat model of Tor. Like all low-latency anonymity networks Tor does not protect against a global passive adversary. Tor has the following adversary model. The adversary can observe some fraction but not the entirety of network traffic. She can generate, modify, delete and delay network traffic and might operate her own relays. Also, the adversary is able to compromise some fraction of the relays in the network [DMS04]. Additionally, in our case the adversary has a considerable amount of resources (in terms of CPU power and bandwidth) at her disposition in order to construct circuits to IPs and mount DoS attacks against specifically targeted onion services.

4.2 Security Goals

We do not want weaken the original security goals of Tor [DMS04], in particular regarding unlinkability. Therefore, we keep the same security goals and add further goals specifically relevant for our DoS defenses. In this section we will describe the crucial security properties of our design. The main contribution of this thesis is the protection of the availability of onion services (against DoS attacks). In order to be useful in the real world it is essential that onion services using our DoS defenses remain easy to use for clients, i.e. the defenses must be intuitively usable and no major performance loss should be introduced in order to mitigate DoS attacks. However, slight inconveniences might be acceptable if they are outweighed by the fact that a user is able to access an onion service even when it is under attack. We point out that a system, that is hard to use, is not only inconvenient but also a threat to anonymity as less users will use such a system (correctly) which effectively reduces the size of the anonymity set.

In [HG11] Henry and Goldberg formalize anonymous blacklisting systems. Their survey paper includes a comprehensive compilation of security properties that these systems should have. We define our own security goals building on top of the paper by Henry and Goldberg. In their definition of an anonymous blacklisting system the authors demand that it must include at least five protocols: registration, token extraction, authentication, revocation and blacklist audit protocol. The objective of those systems is to deny access to malicious users without user deanonymization. Our DoS defenses serve the same purpose. This makes the security properties described in the paper relevant to us, although the

discussed systems consist of different protocols. Registration, revocation and blacklist audit protocols are not included in our own DoS defense system. Therefore, not all of the authors' security properties apply. The following four security goals are our adaptations of the respective goals in [HG11]. We add three more goals goals to complement these goals. We do not describe the security goals formally.

- **Correctness:** The system is considered correct if any authentication token generated (with a valid key), will be accepted by the verifying party, so that the client providing the token is granted access. It is acceptable, if this is not achieved with an absolute guarantee. We accept that a valid token does not check out with a marginal chance, as long as the event is rare enough to be assured it does not affect the usability of the onion service and the token-based access system.
- **Misauthentication Resistance:** An authentication token must only check out if it was generated with the correct secret key and only if that key is still valid. This means that there is no way for a user to circumvent the DoS protection mechanism (investing less resources). Misauthentication resistance leads to the requirement that a malicious client must never learn any information about the secret key, because otherwise clients could issue their own tokens.
- **Backward anonymity:** Given an authentication token from a client and a set of clients with at least two members it must be infeasible to determine which client the token was originally issued to.
- **Unlinkability:** Given two or more authentication tokens it must be infeasible for an adversary to determine if the tokens were issued to the same client. This must hold even if the entity responsible for token issuance and validation is the adversary. Together backward anonymity and unlinkability ensure the client's anonymity.

In their paper the authors of [HG11] also introduce the security goals revocability, revocation auditability and non-frameability. Non-frameability is irrelevant for this thesis since it is only useful if there are trusted third parties that might collude, which is not the case here. Revocability does not apply to our use case, because we do not need to revoke tokens, once they have been issued. Instead, below we introduce limited replayability and limited validity. Because we do not revoke tokens we also do not need to make these revocations auditable for the client. Therefore, we replace revocation auditability with key auditability:

- **Key auditability:** Given an authentication token the client must be able to verify it has been generated using the same key that is used for all clients. This requirement is directly derived from backward anonymity, since without it an attacker might be able to use different keys to issue tokens in order to tag specific clients or at least reduce the size of the anonymity set.
- **Limited replayability:** Given an authentication token a client must only be able to use it for a limited number of authentications with the service.[1] This requirement is necessary, because otherwise an attacker might be able to circumvent the DoS defense mechanism by re-using just one or a small number of tokens. However, we do not require a token to be usable only once as long as the issuing onion service has control to limit the replayability to such a degree, that it does not enable attackers to abuse token replays for DoS attacks.
- **Limited validity:** An authentication token that has once been issued must only be valid for a limited amount of time. The time window during which tokens are valid must be determined by the onion service such that it prevents an attacker from accumulating a large enough amount of tokens in order to carry out a DoS attack with the accumulated tokens. The resulting time window is always a tradeoff between security and usability. Any potential replayability of a token must be taken into consideration, when the length of the time window is determined.

[1] Replaying the same token would, of course, make the client's corresponding actions linkable.

Please note that practical infeasibility suffices to fulfill the security goals, i.e. there may still be a theoretical chance to fail the security properties. That constraint is necessary because of the cryptography used in the DoS defenses, e.g. the cryptographic hash functions allow collisions, which means there cannot be any absolute guarantees.

4.3 Performance Goals

Tor relays, onion services and clients have limited resources. Hence, efficiency is crucial for our design. We define the following performance goals for these three entities.

- **Onion services:** The ability to defend against DoS attacks must outweigh the additional amount of resources invested. In particular the computations required to generate and validate tokens must be significantly less expensive than the computations caused by the DoS attack. Storage, hardware and bandwidth requirements should not increase significantly due to the DoS defenses.
- **Clients:** The onion service must remain usable even for clients running on mobile devices and old hardware. This includes client machines that do not have a GUI. Therefore, the amount of additional computations (for instance from proof of work puzzles), bandwidth and storage should be kept to minimum, as should the user interaction (for instance when solving CAPTCHAs). The design should not increase the number of RTTs required to connect to the onion service, because the connection establishment as defined in the Rendezvous Specification already has a very high latency. In general, the benefit of being able to access the onion service even when under attack should outweigh the additional costs introduced by the DoS defenses for the client.
- **Tor relays:** The quantity of network traffic should not increase significantly due to the DoS defenses. The amount of computations of the involved relays, in particular the IP and RP, should not exceed the point where they can no longer (reliably) provide their services to the network. The storage requirements for those relays should be kept to a minimum. Also, the amount of data stored in the HSDirs should not increase significantly. Therefore, the service descriptor size must only grow within an acceptable range.

Chapter 5

Potential Denial of Service Mitigations

In this chapter we will present three different ideas on how to mitigate DoS attacks against onion services. We will discuss these potential approaches for a DoS defense system and what practical implications they have regarding anonymity and usability. We will not provide exact specifications how these approaches could be realized. This means that we do not pretend to think each idea through with all ramifications. Instead, the objective of this chapter is to give an overview over potential DoS defenses for onion services and their expected implications for security, anonymity, performance and user experience. The reasons for our design decisions as specified in Chapter 6, will become evident throughout this discussion.

Regarding DoS mitigation, one of the major challenges is characteristic for anonymity networks. In anonymity networks there is no trivial solution to identify an attacker and exclusively block her traffic, because the attacker can always acquire a new identity. This attack is called Sybil attack. It is generally desirable to find a solution that does not (entirely) depend on the application layer and is applicable to all onion services, regardless of the protocol they use at the application layer. Additionally, it is our goal to distinguish between cells belonging to DoS attacks and legitimate requests as early as possible. It would be possible to place our defenses at two different points: at the IP and at the onion service. We discard the descriptor level as a potential location for the DoS defense system, although service descriptors already support client authorization. The service descriptor is limited in size and we must avoid frequent changes to it. Once an adversary has gained access to the descriptor it is not possible to defend against the attack dynamically, because it is not possible to identify the attacker. Thus, access to the descriptor could only be revoked by denying access to all clients. Therefore, in the following we will only consider the onion service itself and the IPs as potential point, where illegitimate `INTRODUCE1/2` cells can be discarded.

Ideally, DoS attacks would be blocked at the IP, since that would mean the onion service does not suffer from the attack at all and it would protect an even larger part of the network. No malicious traffic would be forwarded by the IP over the service-side of the introduction circuit. Thus, the attack would not be amplified, which would make it more expensive for the attacker. As explained, the main challenge is the anonymity of the attacker and the lack of information the IP has about the communication between clients and onion service to distinguish legitimate and malicious traffic. In this chapter we will present several ideas to mitigate DoS attacks against onion services. This comparative discussion highlights the reasons for our own design decisions presented in Chapter 6. Table 5.1 provides an overview over the discussed ideas and their respective advantages and disadvantages.

Approach	Advantages	Disadvantages
Proof of Rendezvous: the client retrieves a proof from the RP, which is checked during the introduction phase by the IP or by the onion service	• enforces protocol compliance by the client • increases the costs for a DoS attack • decreases the DoS amplification factor	• no protection of onion services with significantly less resources than attackers • hard to make proofs unforgeable while preserving anonymity • does not make DoS attacks impossible, only makes them harder
Blind RSA Signatures: blind tokens are signed by the onion service, unblinded by the client and validated by the IP in the introduction phase (using a private/ public key pair)	• public key cryptography allows verification at the IP • trivial validation of tokens by the client and unlinkability of tokens • well known non-interactive cryptographic scheme	• public key cryptography is expensive • preventing replay of tokens is (too) hard to enforce in a distributed system with multiple IPs
V-OPRFs: blind tokens are signed by the onion service, unblinded by the client and validated by the onion service in the introduction phase (using one secret key)	• OPRFs based on ECC are computed relatively fast • client verification with DLEQ proofs and unlinkability of tokens • simple non-interactive cryptographic scheme • key rotation is easier than with blind RSA signatures (no certificates needed)	• the onion service has to validate the tokens itself, which must be an extremely efficient operation • client verification is more complex due to DLEQ proofs compared to blind RSA signatures

Table 5.1: Overview over the discussion of DoS defense ideas for onion services.

5.1 Proof of Rendezvous

A crucial factor in the amplification of DoS attacks against onion services is the fact that in the current design an attacker does not have to create a circuit to an RP before sending the `INTRODUCE1` cell to the IP. This means the attacker only has to establish the circuit to an IP in order to cause the victim onion service to try and create another circuit to the specified RP. This scenario gets even worse if the attacker does not rebuild the entire circuit for each `INTRODUCE1` cell, but tears down the last hop of the circuit to the IP and chooses a different relay, so that the IP cannot tell that it is the same circuit and will accept more than one `INTRODUCE1` cell over it. Of course, this makes different `INTRODUCE1` cells linkable, but it might well be possible that a DoS attacker does not care about that.

If we were able to force the attacker to create a circuit to the RP before sending the `INTRODUCE1` cell this would represent a severe increase in the amount of resources required by her to carry out the attack. Therefore, it seems appealing to introduce a proof of rendezvous that is either checked by the IP or the onion service itself before the service creates the rendezvous circuit. In the following we will

discuss how such a proof could work.

In order to maintain Tor's level of anonymity, it is crucial that neither the IP nor the RP can learn each other's identity. Therefore, we consider two options to realize the proof of rendezvous.

1. Once the client has established the rendezvous circuit, the RP creates a signature with a key derived from its identity key including a nonce and a timestamp. The timestamp cannot be the exact time in order to avoid leaking the exact time of the rendezvous circuit construction by the client. However, the timestamp is needed to mitigate replay attacks of the proof of rendezvous. This proof of rendezvous is sent back to the client in the `RENDEZVOUS_ESTABLISHED` cell and the client includes it in the `INTRODUCE2` cell, so it can be checked only by the onion service and not the IP. If the check passes the onion service will construct the circuit to the RP. Otherwise, it will ignore the request avoiding any further consumption of CPU time.

2. The second option is slightly more complex. Again, the RP computes the proof of rendezvous. It is not just a signature but a zero-knowledge proof (ZKP), that proves the client created the circuit to the RP, without revealing which RP was chosen. Please note that it is not the purpose of this section to discuss the cryptographic feasibility of such a scheme, but whether its realization is desirable at all. This proof of rendezvous is sent back to the client in the `RENDEZVOUS_ESTABLISHED` cell and the client includes it in the cleartext part of the `INTRODUCE1` cell, so this time the IP can check the validity of the proof of rendezvous. If the check passes the IP will forward the `INTRODUCE2` cell to the onion service. Otherwise, it will send a `NACK` to the client in the `INTRODUCE_ACK` cell.

The fact that an attacker can create her own relays and use them to precompute fake ZKPs without really creating the rendezvous circuits is an issue for the second option. Currently, RPs do not even have to be part of the network consensus, i.e. clients may specify private relays as RPs and onion services will connect to them.[1] Furthermore, in option 2 the IP is responsible for the verification of the proof of rendezvous. The problem with that architecture is that the IP cannot detect if all the ZKPs were computed using the same attacker controlled relay as an RP. This is the case, because we made it a security requirement, that IP and RP must not learn of each other. Therefore, it is questionable whether option 2 is really suitable to defend against DoS attacks. Additionally, it is not trivial to design a ZKP that matches both the security requirements and has the properties we need to prevent DoS attacks in this scenario. Even if we had such a protocol, a skill- and resourceful attacker might still be able to create her own relays, precompute ZKPs and then mount the DoS attack against the onion service.

Both proposed solutions suffer from one more problem. In the current design, even if the attacker created a rendezvous circuit, she might find a way to make it a less than three hops circuit, because no entity is able to verify the actual length of the circuit. Of course, relays can and do check whether the source of the traffic is a public relay. Nevertheless, the attacker can easily circumvent these checks, if she runs her own relays and uses them as a starting point for the attack. This would mean that an attacker could create rendezvous circuits spending far less computational power for the asymmetric cryptography and path selection required for circuit creation than the onion service that needs to remain anonymous.[2]

The first option has the advantage that the verifier, namely the onion service, learns the RP, because a signature is used instead of a ZKP. So while an attacker can still use her own RP and precompute these signatures, the verifier would at least learn that all the requests it is receiving use the same

[1] This behavior has been discussed, because it was used to make guard discovery attacks easier. Once an onion service's guard node has been discovered, deanonymization of the onion service becomes a lot easier for an adversary. For this attack the attacker would specify her own relay as an RP and the onion service would connect to it [Kra20].

[2] Recent Tor versions allow onion services to create single hop circuits, if the onion service does not care about its own anonymity. The client-side of the protocol is not changed for these services. Only the service uses one hop circuits to connect to the IP and RP. The onion service run by Facebook is an example for such a service.

RPs. This is a strong indication that an attack is going on, because usually the RP should be chosen at random, meaning it is extremely unlikely to see the same RPs over and over. The service could react by temporarily blocking requests that use RPs which are considered malicious because of the bursts in requests using these RPs. Finding appropriate limits for the usage of the same RP is non-trivial as the limits depend on various parameters such as the number of users of the service and the capacity of the relay used as an RP. A disadvantage of the first option is that the check is only performed at the onion service and not the IP. Although no additional circuit has to be created between IP and onion service and the additional network traffic is not considered the bottle neck, this design puts the load of checking the signatures on the onion service. This is certainly an improvement over the creation of a new circuit to an RP, which involves even more CPU time consuming operations. However, checking the proof of rendezvous signatures during a DoS attack might still be enough to exhaust the available CPU time of the service.

Given these considerations we acknowledge that using a proof of rendezvous (at least in the way envisioned here) does not make DoS attacks impossible, but only makes them harder for the attacker, because they need to be more sophisticated. Unfortunately, it is very likely that a motivated resourceful adversary will always find a way around a proof of rendezvous. Therefore, a proof of rendezvous might be introduced to enforce protocol compliance, but it is not enough to defend DoS attacks against onion services.

5.2 Blind RSA Signatures

Blind RSA signatures are cryptographic signatures used in a client server scenario. They allow a client to obtain a signature for a message, without that the server learns the content of the message.

The client has some input x, for which it needs a signature from the server, but the server must not learn the value x. So the client blinds the input x by calculating $x' = r^e x \ (mod\ N)$, where r is a blinding factor, that is relatively prime to the modulus N and is raised to the public exponent e (which is known as part of the signer's public key). It sends x' to the server. The server uses the secret exponent d to sign x' by calculating $z' = x'^d \ (mod\ N)$, and sending z' back to the client. Upon receiving z' the client unblinds it by using the multiplicative inverse of r in $\mathbb{Z}_N$. The client computes $z = r^{-1} z' \ (mod\ N)$. The value z is the plaintext signature for the input x, because $z = r^{-1} z' = r^{-1} x'^d = r^{-1} (r^e x)^d = r^{-1} r x^d = x^d \ (mod\ N)$. The client never learns the server's secret exponent d as part of the private key. Anybody with knowledge of the server's public key k_{pub} is able to verify the signature. Because r is a random number and only known to the client nobody is able to link x' and x. Thus, it is impossible to link signing phase and redemption phase. This scheme has first been introduced in [Cha83; Cha84]. Figure 5.1 shows how blind RSA signatures can be used to issue tokens as part of a DoS system for onion services.

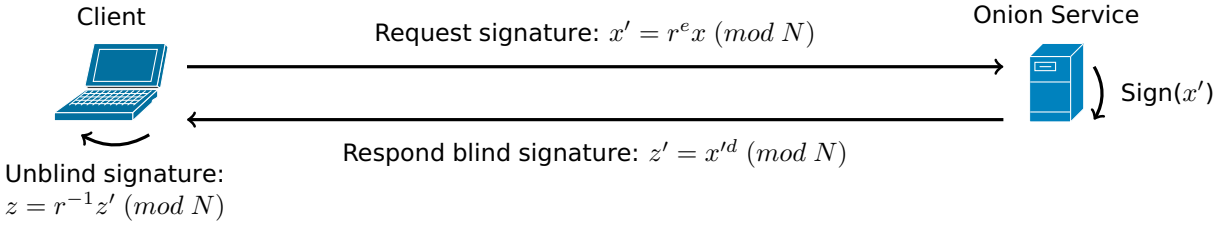

Figure 5.1: The client retrieves a blind RSA signature z' for the input x and unblinds it to obtain the plaintext signature z. The protocol requires 1 RTT.

For our purpose we could use the protocol in the following manner. In order to connect to the onion service a client needs to provide an access token, i.e. a random number x with a corresponding cryptographic signature z provided by the server. This signature is added to the `INTRODUCE1` cell and

checked by the IP. Only if it is valid, the IP will forward the INTRODUCE2 cell to the onion service. Otherwise, it will NACK the INTRODUCE1 cell.

In more detail, this behavior can be achieved as follows. The onion service generates an RSA key pair in order to be used for the generation and verification of blind RSA signatures. The regular onion service key cannot be used, because blind RSA signatures are vulnerable to blinding attacks, where an adversary tricks the server to decrypt a message by signing a blinded message without noticing it. This is possible because in RSA decryption and signing are the same operation. Thus, the key pair must never be used for encryption but for signing only.[3] The onion service upload the public key in the HSDirs as part of the service descriptor. This key pair is refreshed at a given time interval. Every client can use the key in the service descriptor to check that the onion service does not tag users by using different key pairs. When the onion service sends the ESTABLISH_INTRO cells to the chosen IPs it also establishes a rate limit, i.e. how many introductions it is willing to allow in a given time and the onion service includes the public key, so that the IP can verify signatures. If later the INTRODUCE1 cells the IP receives, exceed the limits, it will only forward the requests that contain a valid token to the onion service. The rest of the cells will be discarded.

In the following we describe how a client obtains signatures for its tokens. The client connects to the onion service via an RP following the rendezvous protocol. This is only possible when the service is not under attack, which means the rate limits that the service configured at the IP are not exceeded. The onion service sends a challenge to the client (over the rendezvous circuit). The challenge can either be send and resolved in the application layer or in Tor itself in a relay cell, potentially even in the RENDEZVOUS1 cell. If the challenge is sent, solved and verified over the application layer, the application needs to inform the Tor process that it received a correct solution and that the Tor process may now issue tokens to the respective client. This communication can be realized over the control port of the Tor process. The advantage of handling challenges at the application layer is that they work regardless of which application layer protocol is used. This allows every service provider to use whatever challenges are considered appropriate. After receiving the challenge and solving it the client generates a certain amount of blinded tokens, which it sends to the onion service. If the solution was correct the onion service signs the blinded tokens and sends the signatures back to the client. The client now unblinds the signatures and stores the unblinded tokens together with the unblinded signatures as pairs.

In the future when the client wants to connect to the service it includes the stored pair of token and signature in the plaintext part of the INTRODUCE1 cell, so that the IP can read it. The IP will check the validity of the signature using the onion service's public key. If the check passes, the cell is forwarded to the onion service. Otherwise, the request is denied after a delay.

This scheme has several desirable properties. Signatures are verified at the IP, which is as early as possible. That means it reduces both network traffic and CPU load for the onion service and the network to a minimum. Because of the public key cryptography involved, namely RSA, different keys are used to sign and validate tokens. Thus, the onion service does not have to share a secret key with the IPs. A major disadvantage of this solution is that public key cryptography is computationally expensive. The onion service will need to invest a severe amount of resources in order to generate the blind RSA signatures for the tokens requested. It has to be ensured that requesting a lot of tokens does not lead to a new attack vector. Even if we used a fast blind ECC signature scheme such as the one presented in [JCC10], we would still have to make sure the challenge is hard enough for a potential attacker in order to limit the CPU load of the onion service caused by the signing of blinded tokens.

Another issue of this scheme is a result of the spatial distribution of signing phase and redemption phase between onion service and the IPs. This distribution makes it harder to prevent the replay of tokens. We need to make sure that tokens cannot be spent multiple times. Given that an onion service can create up to 20 IPs and constantly change them, this is not a trivial task. Letting IPs store spent

[3] Another reason, we would need a new key pair for blind RSA signatures is that version 3 onion service identity keys are Ed25519 keys and not RSA keys.

tokens and only accepting unspent tokens might be a solution. This would still allow the attacker the opportunity to spend each token once per IP and reuse it as soon as the onion service changes the IPs. Another option would be to have the spent tokens stored in a shared database by the IPs. This seems an overly complicated solution given that the HSDirs and thus the service descriptors are limited in size and currently there exists no database in Tor that we could use for this purpose. Even with such a distributed database, sophisticated attackers might still have the chance to spend a token simultaneously at all IPs. The obvious way to avoid the problems regarding token replays is to locate both signing and redemption phase at the onion service. Such a protocol could be implemented with OPRFs, as they are more efficient than blind RSA signatures. This idea will be discussed in the next section.

Before discussing OPRF-based tokens in detail, there are two more options worth mentioning. The purpose of both options is to avoid that the onion service itself has to perform all expensive computations. The first option is to generate *and* validate tokens at the IP instead of the onion service. The onion service generates the key pair and passes the public and private key to the IPs in the `ESTABLISH_INTRO` cells. The onion service also uploads the public key in the service descriptor, so that the IPs are committed to use this key for all signatures. The problem is that once an adversary becomes the IP she can start a DoS attack, because she can sign tokens and thus even use the other IPs for the attack. The onion service could stop this attack by changing the key pair and changing the IPs, but this means that no client has valid tokens and therefore the DoS attack could simply be continued the way it is currently realized (without any defenses). A mitigation for this problem is to use a different key pair for each IP. Although this would mean any IP only had to store the tokens spent at this particular IP and there would be no need to share them. It would also bind tokens to a specific IP. This means tokens would only be valid as long as the IP is still used. Furthermore, using different key pairs reduces the size of the anonymity set. We discard both options.

Given these considerations regarding blind RSA signatures it seems to be the most practical solution to accept the fact that an attacker can spend each token once per IP. To avoid this from opening a new attack window the onion service needs to consider it when calculating the number of tokens t a client receives after solving a challenge. Let $n \leq 20$ be the number of IPs and t_{max} be the maximum of rendezvous circuits the onion service is willing to construct per correctly solved challenge. Then, the number of issued tokens is $t = \frac{t_{max}}{n}$. In this scenario the IP only needs to store a token as long as its key pair is still in use. It does not need to share a spent token with anyone, making the protocol a lot simpler than in the ideas discussed above. Of course, this allows a token to be spent again after the service changes to a new IP. This should not be a problem, because publishing a new IP usually also means that one of the former IPs is no longer usable for the DoS attack and the hardness of the challenge ensures that the attacker can only stockpile a limited amount of tokens in a given time. We point out that the amount of tokens an attacker can collect in total is limited by the periodic rotation of the key pair used to generate them. Once the key is rotated, the old tokens become useless and do not pose a threat anymore. For usability of this protocol when determining the time frame one key pair will be in use, it is vital to find a good tradeoff between limiting the number of tokens an attacker can acquire per period and not invalidating tokens too rapidly for legitimate users. Regardless of how long a key pair is valid, the onion service should always have two key pairs in use and frequently replace the oldest key pair (roll over). The latest key pair is used to generate new tokens, while the older one is only used for validation. Thus, not all tokens are invalidated at the same time, when keys are rotated. Instead they can still be used before they are finally invalidated. This ensures that clients always have valid tokens which they can use to connect to the service. It also ensures that clients can use old tokens to access the service and request new tokens even when the service is under attack and rejects cells without valid tokens.

5.3 Verifiable Oblivious Pseudo Random Functions

In this section we discuss an alternative to blind RSA signatures to realize an approach based on cryptographic tokens, which clients receive after solving a challenge. Generally, the idea is very similar to blind RSA signatures as discussed in the previous section. This is going to be the approach that we will fully elaborate in the next chapter, where we present our DoS defense design. The approach uses oblivious pseudo random functions (OPRFs) to blindly sign tokens, which can be used to connect to the onion service. OPRFS can be based on faster cryptography than RSA, such as elliptic curve cryptography. Also, the signatures and tokens are smaller in size. A secure ECC key can be 256 bits, while a secure RSA key should be at least 2048 bits but ideally 4096 bits. The disadvantage is that the cryptography is not asymmetric, which means that there is only a secret key and no public key. The same secret OPRF key is used for token signing and validation. This means the validator has to be a trusted entity. In our case this means the onion service itself has to validate tokens and we cannot upload the secret key in the service descriptor. Therefore, the onion service needs to find another way to prove to the client, that it is not tracking it by using different keys for the OPRF computation. Tokens can either be signed and validated by the onion service or they can be validated by the IP. In the following we discuss both ideas.

The first option is to generate the tokens at the onion service, but validate them at the IP. As in the previous section, the IP will only accept `INTRODUCE1` cells at a certain rate, unless they contain a signed token. The main advantage of this approach is that the attacker's `INTRODUCE2` cells never reach the onion service. Only cells that contain a valid token will be forwarded during a DoS attack. In this design, the onion service does not have to perform any additional computations for token validation. The network also benefits from the approach, because malicious `INTRODUCE1` cells are already discarded at the IP. Token validation would have to be efficient enough, so that IPs are able to perform the operation. This approach would integrate well with Proposal 305, which suggests using rate limits at the IP to protect the entire network [Gou19].[4] While the onion service is not under attack, clients are able to connect to it without tokens, because the rate limits are not exceeded at the IP. During times of high load, clients are able to bypass the rate limits as explained above, if they have obtained valid tokens prior to the DoS attack.

There are several problems regarding this design. The onion service needs to share the secret OPRF key used for token generation and validation with the IPs. If it uses different keys for each IP, tokens and thus users will be bound to specific IPs, which will reduce the size of the anonymity set. This is not acceptable. If the same key is used for all IPs, as explained in Section 5.2, attackers will be able to respend tokens by using different IPs. Also, this increases the risk of suffering a DoS attack, because a malicious IP was picked by the service. In case an IP is malicious it is able to generate its own tokens, because it has access to the secret key, that is also used for signing. Because the same key is used by all IPs, it would not be possible to identify which IP is controlled by the adversary. The onion service could only react by picking a new set of IPs and revoking the secret key for token generation, which would immediately invalidate all tokens legitimate clients obtained earlier. This means in case one malicious IP is picked the adversary is able to start the DoS attack and there are no tokens that legitimate clients could use to bypass the rate limits (and obtain new tokens).

The second option is the following. Tokens are both generated and validated at the onion service using an OPRF. Any `INTRODUCE2` cell containing a valid token will be handled according to the Rendezvous Specification, i.e. a circuit to the RP will be constructed by the onion service. For cells without valid tokens the onion service defines rate limits. In case the service sees more `INTRODUCE2` cells than the rate limits permit it will discard `INTRODUCE2` cells without a valid token. This avoids sharing the secret key with an untrusted third party such as the IP. Legitimate users that have retrieved tokens previously while the onion service was not under attack are always able to connect to the onion service. The main

[4] This proposal was implemented during the time of writing this thesis.

disadvantage of this approach is that all computations are performed by the onion service (and the client). Also, validating tokens at the onion service means that they are only validated at the end of the introduction phase. Ideally, it would be possible to discard malicious cells earlier, i.e. at the IP. Nevertheless, if the cryptographic operations performed by the onion service are efficient enough, this is a viable approach to defend against DoS attacks. Fortunately, we are able to adjust the hardness of the challenge. Thus, the signing operation does not have to be extremely efficient. Token redemption must be efficient enough, so that an attacker cannot exhaust the the onion service's CPU only by sending `INTRODUCE2` cells with invalid tokens.

We summarize the advantages of the second option, i.e. generate and validate tokens at the onion service. The size of the anonymity set is not reduced (under the condition that enough clients adopt the token-based access system). The cryptographic protocol is simpler, it only involves two not three parties and it can be integrated into the existing Rendezvous Specification more easily. However, it is unclear, if it could be combined with Proposal 305, which allows onion services to define rate limits at the IP. With Proposal 305 implemented and activated by the onion service the tokens would be useless when the limits are exceeded at the IP as `INTRODUCE1` cells would be dropped at the IPs so that the tokens would never reach the onion service for validation. Therefore, this scheme and Proposal 305 are mutually exclusive.

OPRFs do not have a public key that the onion service can publish in the service descriptor. Therefore, we need to provide clients with a different way to verify that they are not tagged by the onion service that uses different keys for different clients. Fortunately, there is a solution for that: discrete logarithm equivalence (DLEQ) proofs. A public commitment as part of these proofs can be published in the service descriptor, while a second part is computed dynamically in the token signing phase. The OPRF-based blind signing scheme and DLEQ proofs are everything we need to design a token-based access system for onion services, in order to mitigate DoS attacks. In the next chapter we will present this approach in detail.

Chapter 6

Approach

In this chapter we will present the design of our DoS defenses for onion services. We will first give an overview and then continue with a presentation of our design specification including detailed descriptions of the three different phases of the protocol: setup phase, signing phase and redemption phase.

6.1 Overview

Onion services enable the DoS defenses in the service descriptor by setting the respective flag and providing some additional information such as a public DLEQ verification key. Enabling the defenses also means that the onion service operator configures rate limits for `INTRODUCE2` cells without valid tokens[1] in the `torrc` configuration file. If these limits are exceeded the additional `INTRODUCE2` cells are discarded immediately by the onion service, i.e. without constructing the rendezvous circuit. Clients can bypass the rate limits, if they include a valid token in the `INTRODUCE2` cell. This ensures that the onion service will always stay available to legitimate clients, that have received tokens previously. This holds even when the onion service is under a DoS attack. During such an attack the onion service will not be available to clients that do not have valid tokens. The rate limits ensure that the onion service will not exhaust its CPU by constructing a great number of circuits to RPs. Because tokens are only obtained after solving a challenge, the onion service may specify how many resources an attacker has to spend to get enough tokens for an attack. This makes it practically infeasible to stockpile the required number of tokens. It is crucial that the service spends significantly less CPU time to validate an `INTRODUCE2` cell containing an invalid token than it would need to build the respective circuit to the RP. Otherwise, the use of tokens would not save CPU time, because the attacker could achieve the same effect as before by sending `INTRODUCE2` cells with invalid tokens.

Token signing phase and token redemption phase work as follows. When a client connects to the onion service (for the first time) the onion service provides the client with a challenge. The client solves the challenge and sends the solution back to the service, that verifies the correctness of the solution. Challenges are handled on the application layer and the exact challenge-response protocol is therefore not in the scope of this thesis. Using the application layer to solve challenges ensures that the DoS defenses work regardless of which application layer protocol is used. Please note that in this model Tor is an overlay network on top of TCP/IP. This means the Tor layer is located between the transport layer and the application layer in the TCP/IP stack. If the solution is correct the application layer process uses the Tor control port to inform the Tor process of the correctly solved challenge. The client may

[1] The terms *token* and *signature* may have different meaning depending on the context in which they are used. Both terms are explained in the next section.

now send a certain number of blind tokens, which the onion service will take as an input to an OPRF for which it holds the secret key. The onion service sends the output of the OPRF, the signatures, back to the client. The reply by the onion service also includes a DLEQ proof, which allows the client to verify that the OPRF key corresponding to the public key published in the service descriptor was used by the service. This assures the client that the onion service cannot tag it by using different OPRF keys for different clients, which would make different actions of the same client linkable. The client stores the signed tokens.

In the future the client can include a signed token in the `INTRODUCE2` cell to connect to the service. The token allows the client to bypass any potential rate limits configured by the onion service. This way, legitimate users will always be able to connect to the onion service. The onion service uses a Bloom filter to keep track of spent tokens and prevent clients from using the same token more than once. Because of the cryptographic properties of this protocol token signing and token redemption phase are unlinkable.

As mentioned above the cryptographic scheme used in our design was previously presented similarly in Privacy Pass [Dav+18]. Our own contribution is the modification of Tor to integrate the cryptographic scheme into the Rendezvous Specification. Thus, the scheme itself is not new, but the way we use it to mitigate DoS attacks against onion services is. In the remainder of this chapter we will describe our protocol and the cryptography in more detail.

6.2 Prerequisites

This section describes the prerequisites of the DoS defenses. The security of the OPRF we use is based on the discrete logarithm problem. For performance reasons we use elliptic curve cryptography (ECC). We write all operations in their scalar multiplication form in accordance with elliptic curve notation. However, the protocol should also work with a prime order ring of integers (as used in RSA or Diffie-Hellman key exchange). We will not explain the basics of ECC here, because we make standard assumptions.

Let $\mathbb{G}$ be a cyclic group of prime order q and $X \neq P$ generators of $\mathbb{G}$. Let $Y = kX$ and $Q = kP$ for a secret key k. We call Y the public DLEQ verification key corresponding to the secret OPRF key k. The secret key k is only known by the onion service and drawn uniformly from $\mathbb{Z}_q$. We write: $k \leftarrow \mathbb{Z}_q$. We define $[n] = \{x | x \in \mathbb{N}^+ \wedge x \leq n\}$. H_1, H_2, H_3 and H_4 are hash functions, which are modeled as random oracles. λ is the security parameter. It is the bit length of the random number used as the basis for a token.

$H_1 : \{0,1\}^\lambda \rightarrow \mathbb{G} \setminus \mathcal{O}$, where $\mathcal{O}$ is the identity element of the group $\mathbb{G}$, the point at infinity.
$H_2 : (\{0,1\}^\lambda, \mathbb{G}) \rightarrow \{0,1\}^\lambda$
$H_3 : \mathbb{G}^6 \rightarrow \mathbb{Z}_q$
$H_4 : \{0,1\}^* \rightarrow \mathbb{Z}_q$
$MAC_K(x)$ is a secure message authentication code for an input x, computed using a key K.

Please note, that in this thesis the terms *token* and *signature* are not always used in the same manner for the sake of readability. In Section 6.5 and Section 6.6 the exact steps of the protocol are defined. Thus, it is safe to assume that from the context, in which the terms are used, it becomes clear to what they refer.

Token may have one of the following meanings:

- the random number t, generated by the client at the beginning of the signing phase
- the pair (t, W), where t is the same random number and W is the corresponding unblinded OPRF output, which is also called the signature for t; the pair may also be called *signed token*.

- the pair $(t, MAC_K(req))$, that is used during the redemption phase; the pair may also be called *signed token*.

Signature may refer to:

- the group element W, that is the unblinded OPRF output for the token t
- $MAC_K(req)$, which is computed during the redemption phase and used to prove that the client knows a valid signature W for the token t

6.3 Discrete Logarithm Equivalence Proofs

One building block of our protocol are (batch) DLEQ proofs. They are computed by the onion service during the signing phase. In this section we will explain how these proofs work and why they are important for the security of our protocol. DLEQ proofs are non-interactive zero-knowledge proofs (ZKPs). They allow the client to verify that the onion service has used the secret OPRF key corresponding to the public DLEQ verification key in the service descriptor to sign tokens by checking that $log_X(Y) = log_P(Q)$. Therefore, DLEQ proofs are crucial to achieve *unlinkability* as defined in Section 4.2. In the following we will first describe, how DLEQ proofs work for a single token and then how we can compute DLEQ proofs efficiently for a batch of tokens.

As a precondition, both the onion service and the client need to know X, Y, P, Q. This is not a problem. A consensus about these values has been established in the setup phase and the first part of the signing phase (see Section 6.4 and Section 6.5). The DLEQ proof is computed as follows:

1. The onion service generates a random nonce $s \leftarrow \mathbb{Z}_q$ and computes $A = sX$ and $B = sP$.
2. The onion service calculates the hash $c = H_3(X, Y, P, Q, A, B)$ and $u = (s - ck)\ (mod\ q)$.
3. The onion service sends the pair $D_k = (c, u)$ to the client.
4. The client calculates $A' = uX + cY$ and $B' = uP + cQ$ from the values it received.
5. The client computes $c' = H_3(X, Y, P, Q, A', B')$ and verifies that $c = c'$. If this is the case the client accepts the DLEQ proof. Otherwise, the key k that was used to compute Y and Q is not the same and the blind signature Q will not be accepted by the client.

In step 2 the pre-image resistance of H_3 and the computation of u, which includes a randomly chosen number s ensure, that the client does not learn any information about the secret key k. In step 4 we observe that $A' = uX + cY = (s - ck)X + cY = sX - ckX + cY = A - cY + cY = A$. Furthermore, we observe $B' = uP + cQ = (s - ck)P + cQ = sP - ckP + cQ = B - cQ + cQ = B$. Both equations are only true if if u and c were computed according to protocol and the same OPRF key k was used to compute $Q = kP$ and $Y = kX$. Thus, in step 5, if $c = c'$ the client knows that $log_X(Y) = log_P(Q)$, which means the same key k was used in all of the above operations.

Please note that there is a soundness error of size $1/q$, i.e. there is a negligible probability that the proof is accepted by the verifying client although the onion service used different keys k in the computations of Q and Y. The computation of the DLEQ proof D_k as explained above will be written: $D_k = DLEQ_k(X, Y, P, Q)$.

In the following we describe how a DLEQ proof is computed for a batch of tokens. This saves computation time and network bandwidth, because the DLEQ proof does not have to be computed for each token signature individually, but one DLEQ proof is enough to verify an entire batch of tokens. The DLEQ batch proof has first been presented in [Hen14]. The goal of the batch proof is to allow the onion service to prove to the client $D_{k,i} = DLEQ_k(X, Y, P_i, Q_i)$ for each $i \in [n]$ in an efficient manner. Let n be the number of signed tokens we want to include in the batch proof.

1. The onion service computes $w = H_4(X, Y, \{P_i\}_{i\in[n]}, \{Q_i\}_{i\in[n]})$.
2. The onion service uses w as a seed for a pseudorandom number generator $PRNG : \mathbb{Z}_q \to \mathbb{Z}_q^n$ and receives $c_1, ..., c_n \leftarrow PRNG(w)$.
3. The onion service calculates $M = \sum_{i=1}^{i=n} c_i P_i$ and $Z = \sum_{i=1}^{i=n} c_i Q_i$.
4. The onion service sends $D_k = (c, u) \leftarrow DLEQ_k(X, Y, M, Z)$ to the client.
5. The client knows P_i and Q_i for each $i \in [n]$ and recomputes the first three steps. Therefore, the client also knows M and Z.
6. The client now checks that $log_X(Y) = log_M(Z)$ by using D_k as explained in the algorithm above.

Using the $PRNG$ seeded with a number w, derived from a (pre-image resistant) hash of all values, over which the DLEQ proof is computed, ensures that the onion service cannot choose the OPRF outputs $\{Q_i\}_{i\in[n]}$ such that $Z = kM$ without at the same time fulfilling $Q_i = kP_i$ for each $i \in [n]$. The scalar multiplications c_iP_i and c_iQ_i to derive M and Z ensure, that the order of the signatures Q_i cannot be changed. These scalar multiplications serve as permutations and thus make it infeasible to find values Q_i that yield a valid DLEQ proof D_k without performing the OPRF computation according to protocol: $Q_i = kP_i$.

In step 6, the equivalence of the logarithms assures the client that the same key was used in the OPRF computations, which is crucial to guarantee the *unlinkability* of tokens. We refer to [Hen14] for a proof of the security of (batch) DLEQ proofs. In the following we will denote the DLEQ proof for a key k and a batch of n tokens with: $\bar{D_k} = n\text{-}DLEQ_k(X, Y, \{P_i\}_{i\in[n]}, \{Q_i\}_{i\in[n]})$.

6.4 Setup Phase and Onion Service Descriptors

In order to keep the protocol as simple as possible and make the DLEQ proofs meaningful the onion service has to include the public DLEQ verification keys (and some additional data) in the onion service descriptor. On the one hand, the inclusion in the descriptor will avoid that the use of tokens introduces additional RTTs, i.e. the rendezvous protocol remains non-interactive. On the other hand, the information published in the service descriptor allows the client to verify that it has not been tagged by the onion service by using a different OPRF key in the signing phase than for other clients.

To activate the DoS defenses the onion service sets a flag in the service descriptor. The onion service computes the public DLEQ verification key $Y = kX$ and includes both Y and the generator X in the descriptor, while the OPRF key k remains secret. This assumes that both client and onion service have agreed on a common group $\mathbb{G}$, that is used for the signing and redemption of tokens. The group elements X and Y are published as part of the service descriptor so that the client is assured that all clients receive the same values. Descriptors are fetched anonymously by the client. So even in case of a collusion the same level of security as in the rest of Tor is maintained. Apart from this the service also includes the maximum number of tokens n, which it is willing to issue per correct solution of the challenge. Including n in the descriptor assures the client does not need to retrieve a message from the onion service in order to know how many tokens it may request. This means the token request itself (as it is handled by the Tor process) remains non-interactive. The interactive part of the request, the challenge-response protocol, is handled on the application layer.[2] The onion service periodically rotates the secret OPRF key k. In order to ensure a seamless roll-over of keys the generator X and the public verification key Y will be included *twice* in the descriptor: $\{X_{\text{current}}, Y_{\text{current}}, X_{\text{previous}}, Y_{\text{previous}}\}$. The current key pair is used for signing and validation of new tokens. The key pair from the previous

[2]Theoretically, the parameter in the descriptor could be replaced dynamically during the signing phase, but currently there is no need to do this.

period is only conserved in order to allow clients to have more time to spend their tokens without being required to regenerate them every time the key changes. This roll-over ensures that there is no interruption after the secret key is refreshed. Otherwise, all tokens would be invalidated at once, which would introduce a starting point for a DoS attack.

All of the above parameters and keys are stored in the encrypted section of the service descriptor and uploaded to the HSDirs according to the Rendezvous Specification. Defining the periodicity of key rotation is responsibility of the onion service operator and is therefore not discussed here. Keys should not be rotated too frequently, because key rotations always invalidate old tokens. We decided to allow only two valid keys for token redemption at any given time, because the size of the anonymity set decreases, if more keys are used, as this means less users use tokens signed with the same key k. On the other hand key rotation prevents an adversary from stockpiling a lot of tokens over a long time and then use these tokens all at once in a DoS attack. Such an attack is unlikely, because the number of tokens retrieved per challenge is limited and it would require a high number of correctly solved challenges. Key rotation discourages adversaries from stockpiling tokens, because they will only be valid for a limited time. Another reason to rotate keys is that once a key is invalidated the onion service can forget about all tokens that were already spent based on this key. This decreases the number of spent tokens the service keeps track of and reduces the false positive rate of the Bloom filter, in which those tokens are stored, as explained in Section 6.6.

6.5 Token Signing Phase

The following algorithm describes how clients retrieve tokens. This assumes that the onion service is not under attack, so that the client has been able to connect to the onion service via an RP as described in Section 2.2. Once the connection via the RP is established the client retrieves a challenge from the onion service. The challenge is sent on the application layer, for example as part of a web page. The client solves the challenge and sends the solution back. The application layer server process verifies the solution and if it is correct it uses the Tor control port to inform the Tor process that the client correctly solved the challenge and may now request signatures for tokens. By handling challenges on the application layer we make sure the DoS defenses work regardless of which application layer protocol client and onion service use. Also, this permits the onion service to use challenges such as CAPTCHAs, that require a GUI and user interaction instead of challenges that are only handled without user interaction such as a proof of work. The latter might make the defenses unusable on mobile devices.

Once the Tor process is informed about the accepted challenge the client may send `TOKEN1` cells to request signatures and the onion service will reply by sending `TOKEN2` cells that contain the signatures. `TOKEN1` and `TOKEN2` cells are new types of relay cells. The cell specification is provided in Section 7.4. Figure 6.1 shows how the token signing phase looks on the protocol stack. The number of signatures n the client may request is limited to what the service published in the service descriptor. The cryptographic operations required to sign tokens are depicted in Figure 6.2. Below we describe the protocol step by step. Note that the client and the service may send more than one `TOKEN1` or `TOKEN2` cell per batch. Because the payload of relay cells is limited to 498 bytes, more than one cell might be needed to send the entire batch, depending on the number of tokens requested. Due to the DLEQ proof computation the onion service can only start processing the request once it received all `TOKEN1` cells belonging to the batch.[3]

Let n be the maximum number of tokens n, which a client may request per correctly solved challenge. The client has already retrieved n as part of the service descriptor. The token signing phase

[3] Alternatively, it might be worth discussing to allow optional proof of work challenges, that are handled without involving the application layer. Because these challenges have some disadvantages, particularly regarding mobile devices, we do not elaborate the idea further at this point. It should not be difficult to integrate such challenges into our protocol.

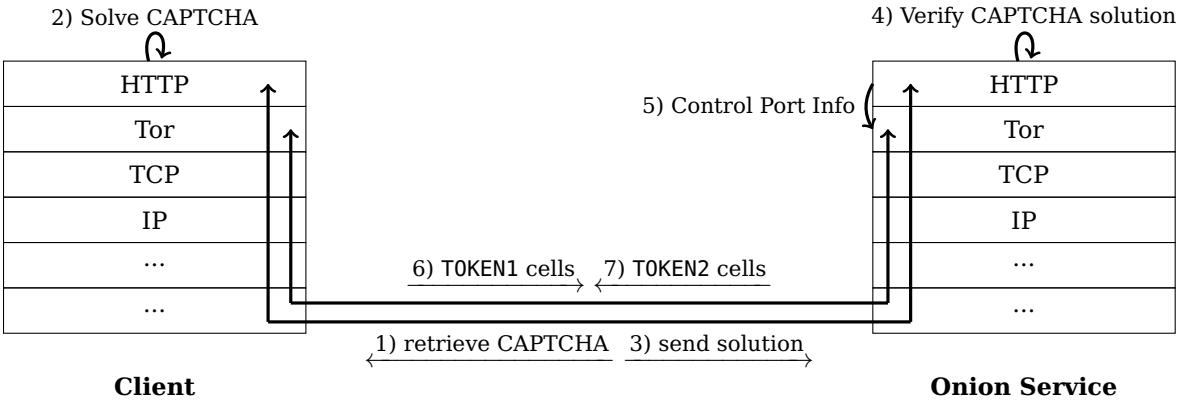

Figure 6.1: In the example a web server is running attached to an onion service and uses CAPTCHAs as challenges. The client retrieves a CAPTCHA from the web server, solves it and sends the solution back. The solution is verified by the web server. The web server informs the Tor process and the client finally requests and retrieves token signatures. The second part of this protocol no longer involves the application layer running on top of Tor and is depicted in more detail in Figure 6.2.

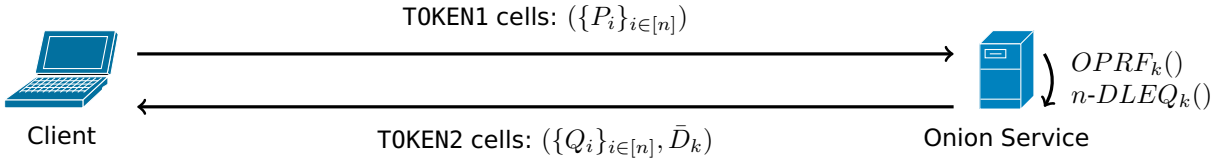

Figure 6.2: The client obtains signatures for its blinded tokens in 1 RTT. The onion service waits until all TOKEN1 cells of the batch have arrived before computing the OPRF output, the DLEQ proof and sending the TOKEN2 cells back to the client.

works as follows.

1. The client calculates n random values of bit-length λ and the corresponding random blinding factors:

   $t_1, ..., t_n \leftarrow \{0,1\}^\lambda$, $r_1, ..., r_n \leftarrow \mathbb{Z}_q$

2. The client computes $T_i = H_1(t_i)$ and $P_i = r_i T_i$ for each $i \in [n]$. The client sends all P_i, with $i \in [n]$ in TOKEN1 cells to the onion service. Combined, the TOKEN1 cells contain the following information: $(\{P_i\}_{i\in[n]})$

3. The onion service computes $Q_i = kP_i$ for each $i \in [n]$ and uses the hash function H_3 to compute the batch DLEQ proof as explained in Section 6.3:

   $\bar{D}_k \leftarrow n\text{-}DLEQ_k(X, Y, \{P_i\}_{i\in[n]}, \{Q_i\}_{i\in[n]})$

4. The onion service sends the signatures for the blinded tokens in TOKEN2 cells to the client. In the first TOKEN2 cell the onion service includes the batch DLEQ proof. Combined, the TOKEN2 cells contain the following information: $(\{Q_i\}_{i\in[n]}, \bar{D}_k)$

5. The client verifies the batch DLEQ proof $\bar{D}_k$ as explained in Section 6.3.

6. The client computes the modular inverses $r_1^{-1}, ..., r_n^{-1}$ of the blinding factors $r_1, ..., r_n$ and uses them to unblind the signatures by calculating $r^{-1}Q_i = kT_i = W_i$ for each $i \in [n]$. The client stores the tokens as pairs (t_i, W_i) for future use.

We summarize the intuition behind the steps of the protocol. In the first two steps the client generates the tokens based on the random numbers t_i and blinds them with the blinding factor r_i, before sending them to the onion service. Computing the hash $T_i = H_1(t_i)$ in step 2 makes sure that T_i is unpredictable for the client. This prevents the client from using just one pair (T_i, W_i) to compute an arbitrary amount of pairs by applying scalar multiplication to both values (sT_i, sW_i). The onion service could not detect this attack due to the hardness of the discrete logarithm problem. The pre-image resistance of the hash function makes sure it is infeasible to find another number t' with $H_1(t') = sT_i$. Therefore, the hash function H_1 is crucial to achieve *limited replayability* as defined in Section 4.2.

The blinding scalar multiplication r_iT_i makes sure that it will be impossible for anyone but the client (that knows the blinding factors r_i) to link a token during signing and redemption phase. The security of this is based on the hardness of the discrete logarithm problem, which makes it practically infeasible to undo the blinding without knowledge of the blinding factor. It also makes it impossible to link two tokens signed in the same batch.

In the steps 3 and 4 the onion service uses the secret key k to compute the OPRF outputs Q_i. The outputs are used to compute the DLEQ proof for the batch of tokens. In step 5 the client verifies the DLEQ proof. If the proof is valid, it was computed using the secret key k corresponding to the public verification key Y published in the service descriptor (with overwhelming probability). Combined with the blinding of tokens, the DLEQ proof ensures *unlinkability* and *backward anonymity*.

In step 6 the client uses the blinding factors to unblind the tokens. It is trivial to compute the modular inverse of the blinding factor r_i^{-1} based on the extended Euclidean algorithm. Because $\mathbb{G}$ is a commutative group[4] the inverse can be used to reverse the blinding of the signature: $r_i^{-1}Q_i = r_i^{-1}kr_iT_i = kr_ir_i^{-1}T_i = kT_i$. Finally, the client stores the unblinded signature W_i and the random number t_i on which the token is based for future redemption.

6.6 Token Redemption Phase

The following algorithm describes how a client uses a signed token to connect to the onion service and bypass the rate limits for `INTRODUCE2` cells set by the onion service. The data flow of the protocol is depicted in Figure 6.3. Below we describe it step by step. Note, that the client may connect to the onion service without including a token in the `INTRODUCE2` cell, but those cells will be discarded if the configured rate limits are exceeded at the onion service. `INTRODUCE2` cells, that contain an invalid token, are treated as cells without a token. Onion service operators must configure the rate limits for each service individually, because the appropriate values vary from service to service.

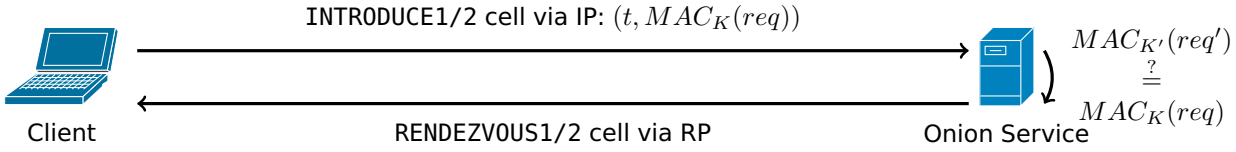

Figure 6.3: The client redeems a token to connect to the onion service. The onion service stores t to prevent double spending. Note, that the figure only shows data and computations relevant for the redemption of tokens not the entire payload of the cells. The request binding data req and req' are onion addresses like: "kq3uyh4ku54fbdgy2ymzkgjdy35y4ebf3ee5snpeknhtmtbc4zii3dyd.onion".

These are the individual steps of the token redemption phase.

1. The client calculates the request binding data req, for which the onion address is used, so that the onion service can easily recompute it.

[4] By definition the group operation is also associative.

2. The client selects a stored token (t, W) and computes a temporary shared key $K = H_2(t, W)$.

3. The client uses the key K to compute $MAC_K(req)$ and sends the pair $(t, MAC_K(req))$ to the onion service. The pair is included in the encrypted part of the `INTRODUCE1` cell. This encrypted part becomes the payload of the `INTRODUCE2` cell after the IP has processed and forwarded the cell.

4. The onion service also calculates the request binding data req' and checks that the token t has not been used previously by any client.

5. The onion service calculates the hash of the token $T' = H_1(t)$, the signature for the token $W' = kT'$ and the shared key $K' = H_2(t, W')$. It uses the key K' to compute $MAC_{K'}(req')$.

6. If $MAC_{K'}(req') = MAC_K(req)$ holds, the onion service accepts the token t and stores it in a Bloom filter. This prevents token replay attacks.

7. The onion service proceeds according to the Rendezvous Specification and constructs the circuit to the RP.

We summarize the intuition behind the steps of the algorithm. In the first three steps the client computes the MAC and sends it to the onion service, along with the random number t on which the token is based. To avoid a cleartext transmission of the shared key K or the signature W, a MAC is used by the client to prove knowledge of a valid signature W. That means a potential passive man-in-the-middle learns neither the signature nor the shared key and thus cannot hijack a token. Hijacking a token would require knowledge of either the blinding factor r or the secret OPRF key k (see [Dav17]). Of course, the `INTRODUCE2` cell is encrypted with the public key of the onion service, so that such an attack should be infeasible, even without these extra measures. Nevertheless, we use a MAC to make the protocol as secure as possible regardless of the context in which it is used. The request binding data in the first step serves as input data for the MAC. It has to be recomputable by the onion service. Theoretically, it could be used to tie a token redemption to a specific resource requested by the client. Currently, this is not supported because in the Rendezvous Specification the client only requests a specific resource once the rendezvous circuit and a TCP stream between client and onion service has been established. In the second step the token t and the unblinded OPRF output W are hashed to derive a shared key K. Thus, knowledge of the key K also proves knowledge of the signature W.

In the steps 4-6 the onion service recomputes the MAC. Knowledge of the random number t is enough to recompute the signature W' and thus the shared key K' and $MAC_{K'}(req')$, because the onion service can recompute the OPRF. In step 6 both $MACs$ will only be equal if the same key is used and the request binding data is the same for onion service and client. If the signature is invalid ($W \neq W'$), the keys used for the MAC will not match and thus the onion service will not accept the token. If the client is able to prove knowledge of a valid signature W for a token t, in step 7 the onion service will construct the circuit to the RP, because it is assured that the request is legitimate.

Please note that the onion service only has to keep track of spent tokens as long as the OPRF key with which they were signed (and validated) is still valid. A Bloom filter is used to do this in an efficient manner. Bloom filters are suitable for this task because onion services with a lot of users otherwise would require a lot of memory to store all tokens (for example in a hash set). In Bloom filters false positives are possible and become more likely the more elements are stored in them. False negatives are not possible. For clients this means that it may happen that a valid token is not accepted because the respective bits are set in the Bloom filter (by tokens that were spent previously). This is acceptable because the client may retry and send a new `INTRODUCE2` cell with another token, that will (most likely) be accepted if the Bloom filter is tuned appropriately to achieve a low false positive rate. Because false negatives do not occur in Bloom filters the attacker cannot spend the same token more than once. We will not discuss how Bloom filters are properly tuned in this thesis, because the number of spent tokens a Bloom filter has to keep track of varies significantly from service to service (due to the number

of users and potentially also due to different deployment scenarios). Privacy Pass has already shown that Bloom filters can be used to store spent tokens in a similar protocol, even for a large CDN like Cloudflare [Dav+18]. Therefore, it is safe to assume that the problem of Bloom filter tuning is solvable. For an example of scalable Bloom filters we refer to [Alm+07].

Chapter 7

Implementation

We have built a prototype of our DoS defenses for onion services in the Tor code base in C.[1] The new functionality both for Tor clients and onion services is added as an extension to the existing code and is backward compatible. We only implemented the DoS defenses for version 3 onion services. We branched off the Tor version `maint-0.4.1`, which was the latest stable release by the time we began this project. In total, we added, deleted or changed 8912 lines of code in 48 different source code files (excluding the additional code required for the evaluation in the respective branch). In the following we will provide details about how we implemented the DoS design. We point out that the primary goal of our implementation was to allow us to evaluate the DoS defenses properly (in the next chapter). Therefore, our code is not secure to use in production. Also, some functionality, that is not required for the evaluation but would be required to make the DoS defenses usable in the real world, has not yet been implemented.[2]

7.1 Cryptography

All cryptographic operations needed for the DoS defenses were implemented by us in a new C module: `hs_dos_crypto.{c,h}`. Because Tor's high level API for elliptic curve cryptography does not provide some of the required functionality we decided to use the OPENSSL library for elliptic curve cryptography and modular arithmetic calculations with big integers. We only used Tor's own cryptographic API for the computation of SHA-256 hashes and HMAC-SHA256. Because OPENSSL also does not provide some functionality for the elliptic curves used by Tor, Curve25519 and Ed25519, we decided to use the NIST curve P-256, also known as `secp256v1`.[3]

Considering the parameters and functions defined in Section 6.2, we choose $\lambda = 256$. This is the bit length of a curve point coordinate of curve P-256 because of the underlying finite field $\mathbb{F}_p$, over which it is defined. Based on this observation, we choose SHA256 as our main hash function and make the hash functions H_1, H_2, H_3 and H_4 variations of it. For the computation of $MAC_K(req)$ we use HMAC-SHA256. To compute the hash H_2 we concatenate the 256 bit token t and the curve point coordinates $W = (x_W, y_W)$ and use it as an input to SHA256: $H_2(t, W) = SHA256(t|x_W|y_W)$. For H_3

[1] The code can be found in our repository in the branches "dos-defenses-master" and "dos-defenses-master-evaluation". The latter branch, by default, is configured to produce data required for benchmarking the DoS defenses. The repository is located at: `https://gitlab.tubit.tu-berlin.de/gaspar_ilom/DoS-defenses-for-onion-services`.

[2] We are in contact with the Tor developer community and are planning on writing a technical proposal based on the results of this thesis in order to add support of the DoS defenses to vanilla Tor.

[3] Usage of this curve has been criticized, because it is feared that its parameter selection was tampered with by the NSA to introduce a backdoor. Regardless of whether this is true or not, Curve25519 has some performance and security advantages over the NIST curve P-256 [Ara+13]. Thus, in production it seems advisable to carefully revise which curve should be used to implement the OPRF.

we concatenate the six input curve point coordinates bytewise and use SHA256 as a hash function. The resulting hash is: $H_3(X, Y, P, Q, A, B) = SHA256(x_X|y_X|x_Y|y_Y|x_P|y_P|x_Q|y_Q|x_A|y_A|x_B|y_B)\ (mod\ q)$. Regarding H_4 we proceed in same way, with the only difference that the input is a variable number of concatenated curve points, because the coordinates of all curve points in $\{P_i\}_{i\in[n]}$ and in $\{Q_i\}_{i\in[n]}$ are part of the input to H_4.

So far we have not mentioned the hash function $H_1 : \{0,1\}^\lambda \rightarrow \mathbb{G} \setminus \mathcal{O}$, defined in Section 6.2. This function hashes a 256 bit random number to a point on the elliptic curve. We provide two different implementations for such a function for the chosen elliptic curve P-256. The first one is called Simplified SWU and has been specified in [Faz+19]. The prototype uses our own C-implementation of Simplified SWU. Simplified SWU is the function, that is also used in Privacy Pass to hash random numbers to a point on the elliptic curve. Because it is defined and explained in the referenced literature we do not discuss it further.

We also implemented a faster alternative to hash random numbers to points on the curve, which we call *Scalar Multiplication with Generator* G (SMG). It is defined as follows $H_1(t) = h(t)G$, where $h : \{0,1\} \rightarrow \mathbb{Z}_q$ is implemented as $h(t) = \text{SHA256}(t)\ (mod\ q)$. This means SMG uses the output of SHA256 to perform a scalar multiplication with the standard generator G of curve P-256 . The result is a point in the cyclic group $\mathbb{G}$ generated by G.

Regarding the security of this procedure [Bri+10] made an important observation. They propose the following construction to hash numbers to points on the curve: $H(m) = f(h_1(t)) + h_2(t)G$ where $h_1 : \{0,1\}^* \rightarrow \mathbb{F}_p$ and $h_2 : \{0,1\}^* \rightarrow \mathbb{Z}_q$ are two hash functions. They argue that the function $H(t)$ is indifferentiable from a random oracle, because $h_2(t)$ is a one time pad an thus yields random results. The reason, they do not simply use $H(t) = h_2(t)G$ to hash to points on the curve is that the discrete logarithm of $H(t)$ would be known, which would make most protocols insecure [Bri+10]. Fortunately, knowledge of the discrete logarithm of $H(t)$ is not a security issue for our protocol. The crucial property for our protocol is the pre-image resistance of H_1. SMG fulfills this property because $h(t) = SHA256(t)\ (mod\ q)$ is pre-image resistant and the scalar multiplication $H_1(t) = h(t)G$ is a permutation, which does not weaken the pre-image resistance (cf. [Bri+10]).

The discrete logarithm of $H(t)$ is known. Without providing a full proof of security, the following considerations give an idea why this is not a security issue. During the token signing phase the client sends $P = rH(t) = rh(t)G$ to the onion service and retrieves the OPRF output $Q = krH(t) = krh(t)G$. During the redemption phase the client sends t (and the $MAC_K(req)$ to the onion service, that may now compute $h(t)$, $h(t)G$ and $kh(t)G$. The onion service must not learn r and the client must not learn k. Due to the hardness of the discrete logarithm problem for elliptic curves, the onion service does not learn r even if it knows $h(t)$, $h(t)G$ and $rh(t)G$. Learning $h(t)$ only allows the onion service to compute rG using the inverse $h(t)^{-1}$. This is not an issue because the service would still have to solve the discrete logarithm problem for elliptic curves to learn r. The client on the other hand for the same reason does not learn k even if it knows $h(t)G$ and $kh(t)G$ or $rh(t)G$ and $kh(t)tG$. The client can use the inverse $h(t)^{-1}$ to compute kG. But this is not an issue as explained above. We conclude for our scheme it is not a problem that the discrete logarithm of $h(t)G$ is known. We have now established that we can use SMG to hash to points on the curve. The evaluation in the next chapter shows that SMG is significantly faster than Simplified SWU. Nevertheless, it is advisable to wait until cryptographers confirm the security of SMG for this use case. This is also the reason why we evaluate the DoS defenses for both hash functions in the next chapter.

7.2 Configuration Parameters

Onion service operators can activate the DoS defenses separately for each onion service and the client by setting the respective parameters in the `torrc` configuration file. The first four of the following parameters are used to configure the DoS defenses at the onion service; the latter two configure the

client:

- `HiddenServiceEnableHsDoSDefense` enables the new DoS defenses. By default these defenses are disabled, which makes the prototype behave indifferentiable from vanilla Tor.
- `HiddenServiceEnableHsDoSDefenseTokenNum` sets the maximum number of tokens a client can request from the onion service per correctly solved challenge, i.e. in one batch request.
- `HiddenServiceEnableHsDoSRatePerSec` sets the maximum rate at which `INTRODUCE2` cells without valid tokens are accepted by the onion service. This does not affect cells containing a valid token.
- `HiddenServiceEnableHsDoSBurstPerSec` sets the maximum burst of `INTRODUCE2` cells without a valid token, that the onion service accepts. This does not affect cells containing a valid token.
- `HsDoSRetrieveTokens` configures the Tor client to request tokens, whenever it has connected to an onion service that has enabled the DoS defenses in its service descriptor.
- `HsDoSClientDir` sets the directory where signed tokens (t, W) are stored. This works similar to the `HiddenServiceDir` parameter, but for the client. If it is not set, tokens are not stored persistently on disk but only in RAM, which means they are lost once the Tor process is terminated.

All of those parameters have minimum, maximum and default values, which have been defined arbitrarily without proper evaluation and therefore should carefully be revised before going into production.

7.3 Onion Service Descriptor

In general, the onion service descriptor and its upload and retrieval to and from the HSDirs remain unchanged. Version 3 of the Rendezvous Specification defines three different layers of encryption for the descriptor. The first one is in plaintext, the second one is encrypted with the unblinded signing key (which means that the HSDirs that do not have knowledge of the onion address cannot decrypt it) and the third layer is encrypted with user authorization keys, that are exchanged out of band and provide access control for the onion service [Tor19]. The only changes we made concern the addition of a flag enabling the DoS defenses (a 4 bytes integer), the required public DLEQ verification keys with the corresponding generators (four elliptic curve points, each of which is 64 bytes in size) and the maximum number of tokens that can be requested per correctly solved challenge (another 4 bytes integer). All of those parameters are added to the third layer of encryption ensuring that only authorized clients can access them (or any client with knowledge of the onion address in case client authorization is not used by the service).

7.4 Cell Definitions and Extensions

We define two new relay cell types `RELAY_COMMAND_TOKEN1` and `RELAY_COMMAND_TOKEN2` cells which are used to send blinded tokens to the onion service and to reply with the the corresponding signatures. For simplicity in the following sections we call these cells `TOKEN1` and `TOKEN2` cells. Furthermore, we define a new cell extension to `INTRODUCE2` cells that allows clients to include a signed token for redemption. We use Trunnel, Tor's standard tool to create secure C code for the parsing of binary data.

Figure 7.1 shows all fields the payloads of `TOKEN1` and `TOKEN2` cells contain. Both cell types contain two boolean flags `first_cell` and `last_cell` (one byte each) that indicate if this cell is the first or last one of a sequence of `TOKEN1/2` cells, related to the same request of a batch of tokens. Both cells have

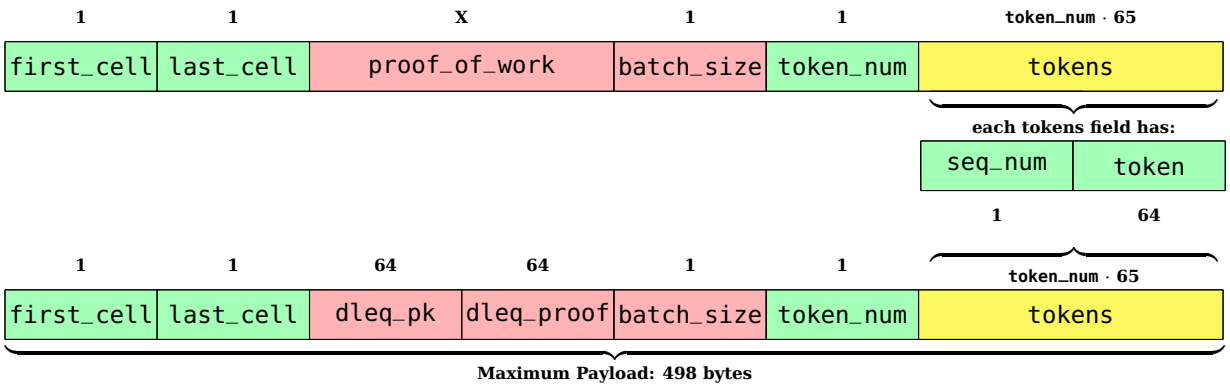

Figure 7.1: The figure shows the payload of a TOKEN1 cell at the top and a TOKEN2 cell at the bottom. The numbers above each field indicate the size of the respective field in bytes. Red fields are only present if the first_cell flag is set. Green fields are mandatory and fixed size. Both cell types contain a variable number of token fields (yellow fields), that contain the blinded OPRF input and output respectively. Each tokens field is composed of two subfields as the figure shows.

another one byte field token_num which holds the number of tokens (TOKEN1 cell) or signatures (TOKEN2 cell) that the cell contains. If first_cell is set, both cell types contain a one byte field batch_size in which the number of tokens in the entire batch is stored, i.e. how many tokens or signatures are sent in this sequence of TOKEN1/2 cells. Both cell types also contain a dynamic cell extension field that allows future addition of parameters to these cells by the definition of a cell extension. Currently, there are no cell extensions defined for these cells. The TOKEN1 cell contains a dynamically sized field proof_of_work which is only available, if the first_cell flag is set. The field is currently unused and only added for forward compatibility. In the future it is suitable to hold a solution for a proof of work, if it is decided that a proof of work challenge can be given in Tor without using the application layer. The last field in the TOKEN1 cell is a dynamic array, named tokens, which holds token_num elements. Each element contains a one byte sequence number (seq_num) and a 64 byte elliptic curve point, the blinded input to the OPRF (token). An identical array also exists in the TOKEN2 cell, with the only difference that the contained curve points are not blinded OPRF inputs but the corresponding OPRF outputs. The sequence number is required by the client to match input and output in case the computed output in the TOKEN2 cells is not in the same order that the inputs were sent. This is not a security issue, because the DLEQ proof would fail, if the client matched input and output incorrectly, but it ensures robustness of the protocol. Furthermore, the TOKEN2 cell contains two more 64 byte fields if the first_cell flag is set. Those fields are dleq_pk and dleq_proof. The first one contains an elliptic curve point, that serves as a public DLEQ verification key and is identical to one of the public DLEQ verification keys the client retrieved in the service descriptor. It is included in the cell, to indicate to the client which of the two keys from the descriptor was used to sign the tokens and compute the DLEQ proof. The second field contains the actual DLEQ proof, that is the pair of 32 byte integers $\bar{D_k} = (c, u)$ with $c, u \in \mathbb{Z}_q$, as specified in Section 6.3.

In order to facilitate token redemption we specify a cell extension called HS_DOS_EXTENSION. It is an extension to the encrypted section of the INTRODUCE1 cell, which becomes the payload of the INTRODUCE2 cell. The extension contains three fields. The first is a 64 byte field pub_key for the public DLEQ verification key that corresponds to the secret OPRF key. This is identical to the key stored in the descriptor, but indicates to the service which of the two keys in the service descriptor is needed to validate the token. The second field is called token and is 32 bytes in size. It holds the random number t, on which the token is based. The last field, redemption_mac, is also 32 bytes in size and stores $MAC_K(req)$ as specified in Section 6.6.

7.5 Unimplemented Features

As mentioned above, we have not implemented all features specified in Chapter 6, but only those needed to build a working prototype for the purpose of an evaluation of the defenses (see next chapter). The most noteworthy features we have not implemented in the prototype are the following.

- **Key rotation:** The onion service does not automatically rotate its secret OPRF keys, although it is able to handle two secret OPRF keys at the same time. More specifically, the onion service handles one current key (used for both issuing and validating tokens) and one previously used key (only used for token signature validation). For the most common deployment scenarios we suggest to rotate keys in the order of weeks. The long periodicity is why an implementation of this feature in the prototype is not useful for our evaluation.
- **Persistent token replay prevention:** The onion service checks that tokens are not spent more than once, but it does not store these tokens persistently on disk. So, if the Tor process is restarted, tokens can be spent again with the prototype.
- **Space efficient token replay prevention:** In the prototype the onion service stores spent tokens in a hash map. Unlike Bloom filters, hash maps do not scale (regarding space) with the number of tokens spent. The space complexity of Bloom filters is $\mathcal{O}(1)$, because classic Bloom filters have a constant size. Hash maps grow with the number of stored entries and have space complexity $\mathcal{O}(n)$.[4]
- **Challenges:** All parts of the DoS defenses concerned with the solution of challenges are not yet implemented. Challenge solution and verification is mainly the responsibility of the application layer. Therefore, we decided to omit it in the prototype. Currently, an onion service, that has the DoS defenses enabled, will always accept token requests from a client (if the request complies with the protocol regarding cell specifications and the number of tokens). Because the challenge-response protocol should be extremely efficient for the onion service, the missing implementation does not affect our evaluation results. [5]
- **Control Port Commands:** The new DoS defenses are not configurable over the Tor control port, but only in the `torrc` configuration file, that is parsed during startup of the Tor process.
- **Tests and security considerations:** We only implemented some very basic unit tests, mostly to test the implementation of the cryptography and parsing of the onion service descriptor. Proper unit tests of the entire added code and integration tests are still a pending task. While the code works fine under normal circumstances, it is not guaranteed to be stable and resilient against attacks. In the current implementation, the prototype does not (always) perform sanity checks for the parameters and occasionally does not validate user inputs properly. Therefore, it probably contains vulnerabilities such as overflows and should only be used for testing and evaluation of the DoS defenses.

[4] Privacy Pass successfully uses a Bloom filter to keep track of spent tokens. Therefore, it is safe to assume that modifying our code to use a Bloom filter is not a difficult problem.

[5] In our `TOKEN1` cell specification we implemented a place holder that makes it easy to handle (proof of work) challenges in Tor itself without relying on the application layer.

Chapter 8

Evaluation

In this chapter we will evaluate our DoS defense system for onion services. We already discussed several advantages and disadvantages of our design decisions throughout the preceding parts of this thesis and in Chapter 6 in particular. We will not repeat all of those considerations. In the first section we analyze if and how the security goals defined in Chapter 4.2 are achieved. In the second section we evaluate the performance of the DoS defenses and in third section we examine the effects of the DoS defenses on network traffic, memory usage and the descriptor size.

8.1 Security Properties

We defined our security goals in Section 4.2. In this section we will evaluate the defenses with respect to these goals. For formal proofs of the security properties we refer to the cited literature. We point out that some design decisions regarding these security properties were already explained in Chapter 6.

- **Correctness:** The correctness of the cryptographic scheme based on the properties of the (verifiable) OPRF which is computed using ECC. The correctness has been formally shown in [JKK14; Jar+16]. If a Bloom filter is used to keep track of spent tokens, correctness is not achieved with an absolute guarantee, because Bloom filters have false positives. Thus, an unspent token might be considered spent by the onion service, because the respective bits were set in the Bloom filter by previously spent tokens. In order to make these limitations on correctness negligible it is crucial to tune the Bloom filter appropriately and keep the false positive rate acceptably low.[1]
- **Misauthentication Resistance:** This property is based on a) the discrete logarithm problem, which ensures the security of the OPRF we use and b) the security properties of the cryptographic hash functions used. Both were proved in [JKK14; Jar+16]. Therefore, we refrain from further elaborations.
- **Backward anonymity:** This property is also based on the discrete logarithm problem that ensures that the blinding of the input can only be undone if the blinding factor is known. Because the blinding factor is random and only known to the client, backward anonymity is achieved.
- **Unlinkability:** Unlinkability is achieved for the same reason as backward anonymity. Additionally, the DLEQ proofs ensure that the onion service cannot distinguish which client a token was issued to, because the same OPRF key is used for all clients during signing and redemption phase. For a proof of the security see [JKK14; Jar+16].

[1] We already discussed the importance of correct tuning of Bloom filters in Section 6.6.

- **Key auditability:** As stated above, this property is achieved due to the security of DLEQ proofs that allow the client to verify if the key that was uploaded in the public service descriptor is the same as the key that was used to compute the signatures. Since the service descriptor is downloaded anonymously (over a Tor circuit), the same security properties as for the rest of Tor ensure the client that it cannot be tagged by the onion service, even if the onion service colludes with the HSDirs. This is the case, because the HSDirs cannot distinguish different clients and thus cannot respond with a tailor-made descriptor in order to tag a specific client by using a custom key for the OPRF computation.
- **Limited replayability:** The onion service uses a Bloom filter to keep track of spent tokens. With Bloom filters, false negatives are impossible [Alm+07]. This ensures that each token can only be spent once.[2]
- **Limited validity:** This property directly follows because we achieve misauthentication resistance and rotate keys. Once the key that was used to issue the token is no longer valid, the token will not check out. Only tokens issued with a valid key can be redeemed as misauthentication resistance ensures.

We have shown informally and based on the cited formal proofs, that the protocol we designed has all desired security properties.

8.2 Performance

In this section we evaluate the performance of the DoS defenses and show that the performance goals defined in Section 4.3 are achieved. We begin the performance evaluation with a comparative analysis of the CPU time consumption of the signature validation and rendezvous circuit construction. The ratio between both operations is crucial to determine how much CPU time the onion services effectively saves because of the token-based DoS defenses. In a second part of this section we provide benchmarks for all cryptographic operations (for both client and onion service). All data for the performance evaluation was collected on the following hardware: a Desktop computer with 64 bit AMD Ryzen 3 1200 Quad-Core Processor with 3100MHz with 16GB of DDR-4 RAM. We ran our tests with Tor running on Ubuntu 18.04. This setup was used to generate the benchmarks for both client and onion service.

The DoS attack we aim to mitigate is based on CPU time exhaustion of the targeted onion service. This means in order for the DoS defenses to work efficiently it is crucial to prevent an adversary from using up too much of the service's limited CPU time. For the evaluation we measure and compare the CPU time that is consumed when constructing a circuit to an RP and when validating an invalid token obtained in the payload of an `INTRODUCE2` cell. We compare these two operations because an `INTRODUCE2` cell with an invalid token maximizes the amount of computations needed to handle the cell for the onion service. Depending on which hash algorithm is used for token generation, the token validation operation either requires several expensive calculations with big numbers (SSWU) or a scalar multiplication on an elliptic curve (SMG). Of course, this assumes that the attacker cannot get hold of (enough) valid tokens. This assumption is reasonable if the service operators configure sufficiently hard challenges for clients to obtain tokens. `INTRODUCE2` cells without tokens can be discarded easily by the onion service and therefore do not cause a significant CPU workload. Hence, token validation must be significantly less expensive in terms of the CPU time that is required compared to the construction of a circuit to an RP. Otherwise, an attacker would only have to modify the attack slightly and add invalid tokens to her `INTRODUCE2` cells.

[2] Alternatively, other mechanisms such as hash sets might be used to keep track of tokens, as we do in the implemented prototype. Hash sets neither have false negatives nor false positives, but they might not be space efficient enough for large onion services, which makes Bloom filters the preferable mechanism for services that expect to keep track of a large number of spent tokens.

It is not trivial to measure the CPU time needed for the onion service to build a circuit to the RP, because this happens in several steps. Some of those steps depend on the reactions of relays chosen for the path of the respective circuit. In theory, the most expensive operations of the circuit construction should be the path selection algorithm, the cryptographic handshake with the client and the handshakes with all four relays that form part of the circuit (including the RP itself). Our benchmarks show that 92% of the CPU time spent during the construction of a rendezvous circuit goes into the path selection algorithm. Therefore, an optimization of this algorithm might also help defending against DoS attacks. However, changes to the path selection algorithm may have security ramifications because a new path selection algorithm might introduce path biases that make deanonymization attacks easier. We do not advise to modify the path selection prematurely.

Operation	**SSWU**		**SMG**	
	Mean	STD	Mean	STD
Rendezvous Circuit Construction	6416	1019	6097	942
Token Validation	134	5	98	5
Ratio between both Operations	47.8	-	62.4	-

Table 8.1: The table shows the average CPU time in µs the onion service spends in the rendezvous circuit construction and for token validation. The measurements are provided for both hash functions SSWU and SMG. The sample size for both hash functions is 1000. The standard deviation (STD) of the rendezvous circuit construction time is caused by the time variance of the path selection algorithm. It is not an effect of which hash function is used.

For the comparison between the CPU time that is spent during rendezvous circuit construction and token validation, we ran an onion service in the real Tor network and used a modified client in order to send 1000 `INTRODUCE2` cells to the onion service. We measured the CPU time spent in both operations (token validation and rendezvous circuit construction) including not just cryptographic but all operations, such as the parsing of the cells. We ran this experiment with both `hash_to_curve` functions: SSWU and SMG. The results in Table 8.1 show that token validation requires far less CPU time (regardless of which `hash_to_curve` function is used) than the construction of a rendezvous circuit. Using SSWU to hash random numbers to curve points, token validation is 47.8 times faster than rendezvous circuit construction and using SMG it is 62.4 times faster. Both results support the notion that our defenses will make DoS attacks much harder for attackers.

Of course, these factors (47.8 and 62.4 respectively) do not immediately mean that DoS attacks become harder by the same factor as soon as the DoS defenses are enabled, i.e. the quantity of required `INTRODUCE1` cells does not immediately increase by the same factor. This is not the case, because what we measured is not the only CPU time consuming procedure in the Tor process. In addition to the `INTRODUCE2` cells with invalid tokens sent by the attacker, there are also the cells from legitimate clients that contain tokens and potentially also some cells that do not contain tokens but are accepted because the rate limits defined by the service were not yet exceeded. All of those cells will trigger rendezvous circuit constructions consuming CPU time. Because the rate limits depend on the deployment scenario we cannot name an exact factor by which the costs for a DoS attack increase after enabling the DoS defenses. Instead, we assume that adequate parameter selection and wide adoption of the DoS defenses by clients should allow the service to get close to the factors we measured. Increasing the number of `INTRODUCE2` cells required for a DoS attack by more than 40 (or potentially more than 60) times should be enough to protect most onion services from DoS attacks. However, this result has yet to be confirmed by real world onion services.

In addition to the comparison between rendezvous circuit construction and token validation we

Entity	Operation	Computation Time			
		Absolute Time		Relative Time	
		SSWU	SMG	SSWU	SMG
Client	Token Generation (incl. Blinding)	3123	2204	104	73
	Unblinding Signatures and DLEQ Proof Verification	4156	4153	139	138
	Redeem 1 Token	8	8	8	8
Onion Service	Token Signature (OPRF and DLEQ Proof Computation)	3482	3482	116	116
	Validate 1 Token	114	80	114	80

Table 8.2: This table provides benchmarks for all cryptographic operations of the DoS defenses. The batch size is the current default value: 30 tokens. Times are in µs. The times shown are the mean computed over a sample of size 1000. *Absolute Time* refers to the time for the entire batch. *Relative Time* refers to the time required per token. The table provides benchmarks for both hash functions SSWU and our custom hash function SMG. The redemption and validation operations only process single tokens.

provide performance benchmarks for all relevant operations of both client and onion service during token signing and redemption phase. Table 8.2 shows the results of those benchmarks. We measure the validation time for both `hash_to_curve` functions (SSWU and SMG) specified in Section 7.1. As expected, our results show that SSWU is slower than SMG. The benchmarks in Table 8.2 and Figure 8.1 and Figure 8.2 only measure the time spent for the cryptographic operations and not in other related operations such as parsing `INTRODUCE2` cells. All of these figures and the table are based on the same data.

The results in Table 8.2 show that all operations run extremely fast and should not cause issues in a regular deployment. We observe that the client can redeem stored tokens almost without spending any CPU time and that all other operations are in a similar time range, if we consider CPU time consumption per token. This means that on average both client and onion service spend a similar amount of CPU time during an entire life time cycle of a token. Given that the onion service may force the client to spend an arbitrary amount of resources to find a solution to the challenge during the signing phase this is a good result.

From the table we can immediately derive that using SMG to hash random numbers to a point on the elliptic curve requires only 70.6% CPU time for token generation by the client (with batch size 30 tokens) and 70.2% CPU time for token validation by the onion service (for a single token) compared to the CPU time required with SSWU as `hash_to_curve` function. While the increase in performance for the client is not relevant, the improved performance of the token validation is vital to our DoS defenses. Thus, comparing token validation to the CPU time required from the onion service for a rendezvous circuit construction, the token validation with SMG is even more efficient than with SSWU. As explained above, this means that the onion service is able to handle a larger amount of `INTRODUCE2` cells with invalid tokens in case an adversary mounts such an attack.

Figure 8.1 shows how the spent CPU time increases with the number of tokens in a batch and Figure 8.2 shows that the time spent per token is almost constant for each operation. The CPU time required grows linear with the number of tokens in the batch. The figure also shows that there is a fixed time overhead due to the computation and verification of DLEQ proofs for the respective operations. This overhead is already hardly perceivable for a batch size of 10 tokens, as the figure indicates.

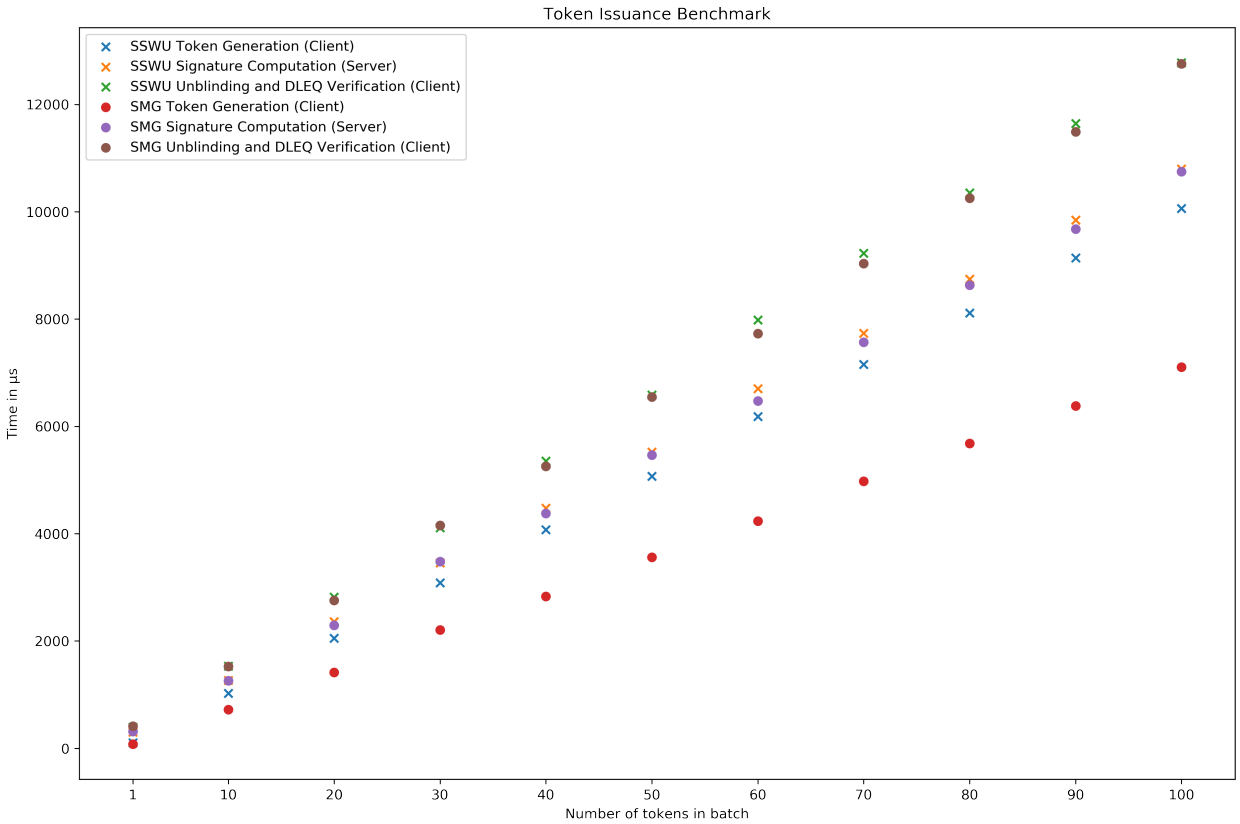

Figure 8.1: The graph shows that (apart from a constant time required for the computation or verification of the DLEQ proof) the time required for each operation grows linear with the number of tokens. The difference between the hash functions used becomes most notable for the token generation.

8.3 Network Traffic, Memory Usage and Descriptor Size

Regarding network traffic and RAM usage the DoS defenses should not have any noticeable impact, because tokens are only stored in RAM for a very short time. The size of the `INTRODUCE1` and `INTRODUCE2` cell does not change if a token is redeemed because the cell has a fixed size of 512 bytes, that are mostly padding. Token requests on the other hand should not occur very frequently, but even for a batch of 100 tokens, only 15 `TOKEN1` and `TOKEN2` cells are required. This is 7680 bytes of network traffic per direction and should not be relevant under normal circumstances. It follows directly, that the DoS defenses do not affect Tor relays significantly, including those on the rendezvous and introduction circuits. The only change from the Tor relays' perspective are the `TOKEN1` and `TOKEN2` cells (and the challenge and solution sent over the application layer) that are routed through them. Compared to the amount of generic application layer traffic that is to be expected, this does not represent a relevant increase of network traffic. Note that the code that is executed by relays is not changed, because all changes only concern the onion service and the client.

The redemption of tokens is non-interactive and requires less than one RTT, because the client only has to add the token to the `INTRODUCE2` cell and send it to the onion service (via the IP). This means that once a clients has valid tokens the DoS defenses do not add additional latency. In the signing phase two additional RTTs are required, one RTT for obtaining and resolving the challenge over the application layer and one RTT for sending the `TOKEN1` and replying with `TOKEN2` cells. Fortunately, this happens simultaneously to the application layer traffic that is exchanged between client and onion

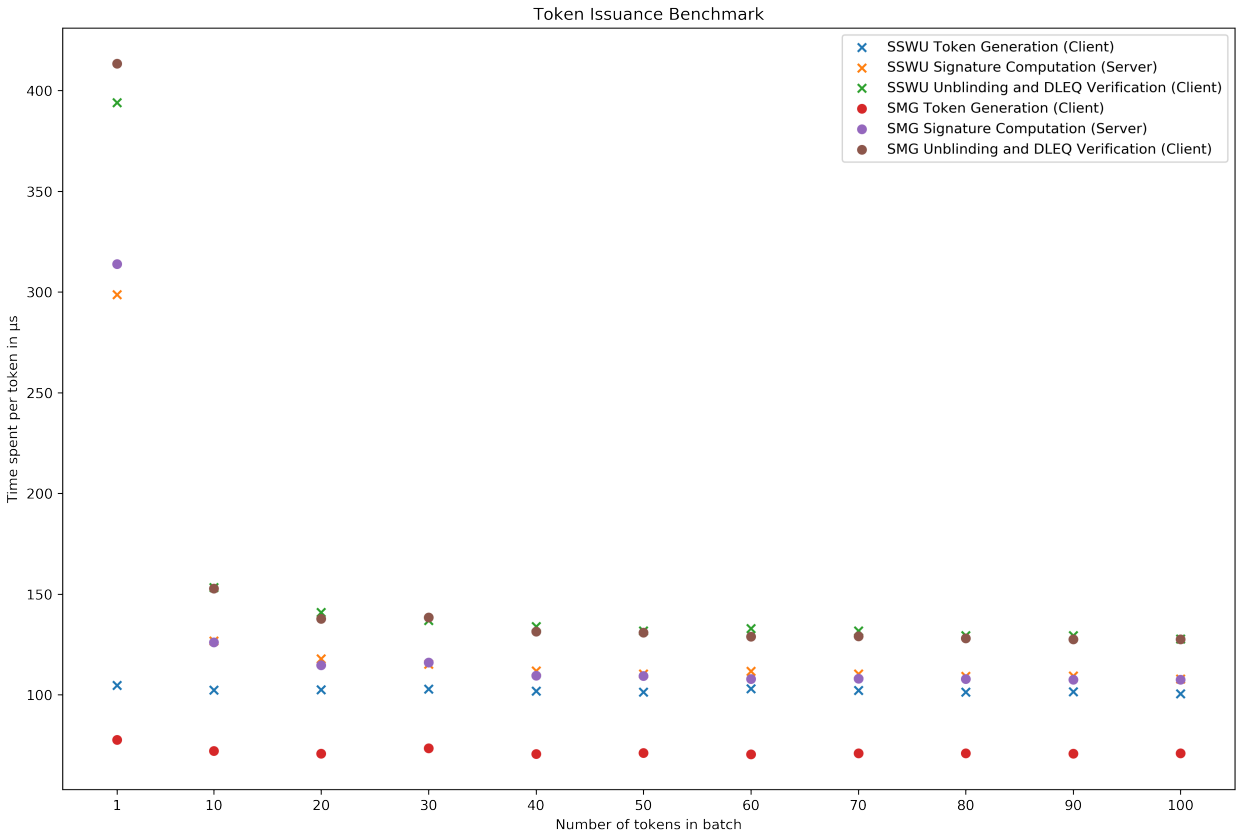

Figure 8.2: The graph shows that the CPU time spent per token is almost constant for each operation. There is some overhead due to the CPU time spent for DLEQ proof computation and verification in the respective operations.

service. Therefore, the signing phase also does not affect the regular user experience (apart from the challenge which users have to solve every time a batch of tokens is requested).

Regarding memory usage we are mostly concerned about the onion service. Fortunately, the service can forget about signed tokens because only the client needs to store signatures and tokens to spend them. The onion service can keep track of spent tokens in a space efficient manner by using a Bloom filter. The tuning of the Bloom filter is not part of this thesis because Privacy Pass has already shown that Bloom filters are suitable to store spent tokens [Dav+18].

The only potential problem regarding memory usage are unfinished token requests. If the client announces a batch size of 100 tokens but only sends 99 tokens and never sends the final `TOKEN1` cell containing the last token, the onion service has to keep the 99 tokens it already received for signing in its memory. One token is only 64 bytes in size with relatively little management overhead per batch. Given that an attacker would have to solve one challenge to fill a maximum of 650 bytes of memory, this is not a viable way to force the service to run out of memory. The attack can easily be mitigated by the onion service by adding a timeout for initiated token requests. That means, if the client does not send the entire batch within a certain time, the onion service will free the memory. Note that this feature is not implemented in our prototype, which is not a problem, because the memory is freed as soon as the rendezvous circuit is torn down. This happens after a timeout.

Regarding the size of the onion service descriptor we make the following observation. The entire onion service descriptor has a maximum size of 50kb. Given that it is Base64 encoded and encrypted

and also requires a descriptor header and footer the maximum size for the superencrypted second layer is 30kb. Since most onion service descriptors have a superencrypted block of about 10kb it is desirable to aim for that size, because it maximizes the anonymity set. Fortunately, this is not a problem. If the descriptor contains three IPs and 16 authorized clients, there are about 2kb of padding, in which the parameters can easily fit. Even if we consider that tags are added to facilitate parameter parsing and that the parameters are encrypted and Base64 encoded, the data added to the third layer is less than 750 bytes. Although this slightly increases the descriptor size, given that there usually is unused 0-padded space, the increased size of the third layer should have no practical effect on the descriptor size in all common use cases. For some further calculations regarding the sizes of the different layers in the service descriptor see [Kad16].

Chapter 9

Conclusion

In this thesis we presented a new DoS defenses subsystem for Tor onion services. Hitherto, onion services would be vulnerable to DoS attacks if the attacker was able to trigger enough rendezvous circuit constructions, because (rendezvous) circuit constructions consumed a relatively high quantity of CPU time. We defend against this attack by adapting the cryptographic scheme introduced in Privacy Pass [Dav+18] and integrating it into Tor as an extension to the Tor Rendezvous Specification. The approach we use is to define rate limits for `INTRODUCE2` cells at the onion service, so that any `INTRODUCE2` cells exceeding these limits are discarded by the service. To guarantee the availability of the onion service to legitimate clients, these clients can include cryptographic access tokens in their `INTRODUCE2` cells and in this way bypass the rate limits.

The protocol is based on a verifiable OPRF which the onion service uses both for signing of blinded tokens and for token validation. Because the onion service configures rate limits for `INTRODUCE2` cells Clients can request a limited number of these service-bound tokens from the onion service after solving a challenge retrieved from the service over the application layer. Handling challenges on the application layer ensures that appropriate types of challenges (such as CAPTCHAs for websites) can be used regardless of which application layer protocol is used. By limiting the acceptance of `INTRODUCE2` cells to a certain rate onion services operators are able to avoid CPU exhaustion caused by the construction of rendezvous circuits. At the same time, clients that have previously retrieved a signature for a token from the onion service are able to spend the token, in order to access the service (and obtain new tokens), even when the onion service is throttling traffic because the rate limits are exceeded. The protocol ensures that onion services remain available to legitimate users, at least if those users have accessed the service before and retrieved a signature for a token. Clients may obtain multiple signed tokens for just one correct solution of the challenge. No user interaction is required except for the challenge (in case the onion service uses an interactive challenge, such as a CAPTCHA). Because tokens ensure that challenges only need to be resolved from time to time our design reduces the effects on user experience to a minimum. The original security and anonymity guarantees of Tor are not weakened by the DoS defenses because the OPRF-based blind signature scheme, combined with DLEQ proofs, cryptographically ensures that token signing and token redemption are unlinkable. Furthermore, two tokens obtained in the same batch are also unlinkable and thus cannot be traced back to the same client. Therefore, our DoS defenses protect the availability of onion services without deteriorating the security or user experience.

We will briefly summarize the results of this thesis. After the introductory chapter, Chapter 2 provided an overview over the Tor protocol and a short introduction to version 3 of the Tor Rendezvous Specification. The problem statement explained why it is so easy for an attacker to carry out a DoS attack by exhausting the CPU of the targeted onion service. Chapter 3 discussed related work on DoS attacks in Tor and how to detect them. The chapter also included a section on the cryptography re-

quired by the DoS defenses. In Chapter 4 we continued the thesis with a definition of our threat model and the security goals of our design. Chapter 5 discussed three different potential DoS mitigations. This comparative analysis explained the reasoning behind our design decisions and pointed out the advantages of the cryptographic scheme the DoS defenses are based on. The design of the DoS defenses and an exact specification of the cryptographic schemes used were presented in Chapter 6. Chapter 7 explained the details of the implementation of the prototype of our DoS defenses. This chapter described how our approach was integrated into the Rendezvous Specification and which new cell formats were defined. In this chapter we also enhanced the cryptographic scheme with the presentation of a new faster function, that hashes numbers to points on the NIST elliptic curve P-256. We evaluated the DoS defenses in Chapter 8.

For the evaluation we used the implemented prototype to run performance benchmarks for our protocol. We analyzed theoretically how the defenses have changed the optimal potential DoS attack. The benchmarks show that the protocol runs fast enough at both client and onion service for token issuance and redemption. It is unlikely that the DoS defenses will have any noticeable impact on performance under normal circumstances. If the onion service correctly configures the hardness of the challenge that must be solved in order to retrieve token signatures, the most efficient attack now is flooding the service with `INTRODUCE2` cells that contain invalid tokens. Our evaluation shows that the token validation operation runs 47.8 (SSWU) and 62.4 (SMG) times faster than the RP circuit construction, depending on which function is used to hash numbers to points on the elliptic curve. If an attacker carries out the attack with invalid tokens a successful attack will require almost 47.8 (or 62.4 respectively) times more `INTRODUCE2` cells than before. Of course, this is only true if the service operator correctly configured the DoS defenses parameters and set an appropriately hard challenge. Therefore, the actual improvement due to our defenses may vary depending on the deployment scenario.

Concluding, we recognize that anonymity networks represent unique challenges regarding sanctioning bad user behavior. In anonymity networks, by definition, different events should not be linkable to a single user or to one another. This makes it easy for an adversary to carry out attacks without the risk of being identified. Anonymity also means that attackers cannot permanently be excluded from using a free-to-use anonymity network like Tor. Nevertheless, we showed that it is possible to mitigate DoS attacks in this scenario by using a cryptographic blind signature scheme based on OPRFs. Future research and practical use of our mitigations will show whether our DoS defenses perform fast enough to discourage attackers from mounting CPU-based DoS attacks against onion services. As attackers will most likely try to find other ways around our DoS defenses, the Tor community will have to continue incrementally adapting and improving the DoS defenses in order to provide anonymous communication to its users.

Bibliography

[Alm+07] Paulo Sérgio Almeida et al. “Scalable Bloom Filters”. In: *Information Processing Letters* 101.6 (2007).

[Ara+13] Diego F Aranha et al. “A note on high-security general-purpose elliptic curves”. In: *IACR Cryptology ePrint Archive* 2013 (2013).

[Art15] Article 19. *Right to Online Anonymity*. 2015. URL: https://www.article19.org/data/files/medialibrary/38006/Anonymity_and_encryption_report_A5_final-web.pdf (visited on 02/14/2020).

[Bac97] Adam Back. *Hashcash*. 1997. URL: http://www.cypherspace.org/hashcash/ (visited on 09/21/2019).

[Boc+19] Cecylia Bocovich et al. *Addressing Denial of Service Attacks on Free and Open Communication on the Internet: Final Report*. Tech. rep. 2019-05-001. The Tor Project, 2019. URL: https://research.torproject.org/techreports/dos-censorship-report2-2019-05-31.pdf (visited on 02/14/2020).

[Bor+07] Nikita Borisov et al. “Denial of Service or Denial of Security?” In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS ’07. ACM, 2007.

[Bri+10] Eric Brier et al. “Efficient Indifferentiable Hashing into Ordinary Elliptic Curves”. In: *Advances in Cryptology – CRYPTO 2010*. Springer, 2010.

[Cha83] David Chaum. “Blind Signatures for Untraceable Payments”. In: *Advances in Cryptology – CRYPTO ’82*. Springer, 1983.

[Cha84] David Chaum. “Blind Signature System”. In: *Advances in Cryptology – CRYPTO ’83*. Springer, 1984.

[CMZ14] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. “Algebraic MACs and Keyed-Verification Anonymous Credentials”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Association for Computing Machinery, 2014.

[CP93] David Chaum and Torben Pryds Pedersen. “Wallet Databases with Observers”. In: *Advances in Cryptology – CRYPTO ’92*. Springer, 1993.

[CR19] Geoffroy Couteau and Michael Reichle. “Non-interactive Keyed-Verification Anonymous Credentials”. In: *Public-Key Cryptography – PKC 2019*. Springer, 2019.

[Dan+12] Norman Danner et al. “Effectiveness and Detection of Denial-of-service Attacks in Tor”. In: *ACM Trans. Inf. Syst. Secur.* 15.3 (2012).

[Dav+18] Alex Davidson et al. “Privacy pass: Bypassing internet challenges anonymously”. In: *Proceedings on Privacy Enhancing Technologies* 2018.3 (2018).

[Dav17] Alex Davidson. *Privacy Pass - “The Math”*. The Cloudflare Blog. 2017. URL: https://blog.cloudflare.com/privacy-pass-the-math/ (visited on 09/15/2019).

[DKL09] Norman Danner, Danny Krizanc, and Marc Liberatore. “Detecting Denial of Service Attacks in Tor”. In: *Financial Cryptography and Data Security*. Springer, 2009.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. “Tor: The Second-Generation Onion Router”. In: *13th Usenix Security Symposium*. USENIX, 2004.

[Faz+19] Armando Faz-Hernández et al. *Hashing to Elliptic Curves. draft-irtf-cfrg-hash-to-curve-05*. Internet Draft. Work in Progess. 2019. URL: `https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-05/` (visited on 11/15/2019).

[FHS15] Georg Fuchsbauer, Christian Hanser, and Daniel Slamanig. “Practical Round-Optimal Blind Signatures in the Standard Model”. In: *Advances in Cryptology – CRYPTO 2015*. Springer, 2015.

[Fra+07] Nicholas Fraser et al. “Using Client Puzzles to Mitigate Distributed Denial of Service Attacks in the Tor Anonymous Routing Environment”. In: *2007 IEEE International Conference on Communications*. IEEE Computer Society, 2007.

[Gar+11] Sanjam Garg et al. “Round Optimal Blind Signatures”. In: *Advances in Cryptology – CRYPTO 2011*. Springer, 2011.

[GG14] Sanjam Garg and Divya Gupta. “Efficient Round Optimal Blind Signatures”. In: *Advances in Cryptology – EUROCRYPT 2014*. Springer, 2014.

[Gou19] David Goulet. *Proposal 305: ESTABLISH_INTRO Cell DoS Defense Extension*. Tor Developer Mailing List. 2019. URL: `https://lists.torproject.org/pipermail/tor-dev/2019-June/013875.html` (visited on 01/14/2020).

[Hen14] Ryan Henry. *Efficient Zero-Knowledge Proofs and Applications*. UWSpace. 2014. URL: `http://hdl.handle.net/10012/8621` (visited on 02/14/2020).

[HG11] Ryan Henry and Ian Goldberg. “Formalizing Anonymous Blacklisting Systems”. In: *2011 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2011.

[Jan+14] Rob Jansen et al. “The sniper attack: Anonymously deanonymizing and disabling the Tor network”. In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS ’14)*. Internet Society, 2014.

[Jar+16] Stanislaw Jarecki et al. “Highly-Efficient and Composable Password-Protected Secret Sharing (Or: How to Protect Your Bitcoin Wallet Online)”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, 2016.

[JCC10] Fuh-Gwo Jeng, Tzer-Long Chen, and Tzer-Shyong Chen. “An ECC-based blind signature scheme”. In: *Journal of Networks* 5.8 (2010).

[JKK14] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. “Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model”. In: *Advances in Cryptology – ASIACRYPT 2014*. Springer, 2014.

[Kad16] George Kadianakis. *Sketch: An alternative prop224 authentication mechanism based on curve25519*. Tor Developer Mailing List. 2016. URL: `https://lists.torproject.org/pipermail/tor-dev/2016-November/011658.html` (visited on 01/19/2020).

[Kad19] George Kadianakis. *Denial of service defences for onion services*. Tor Developer Mailing List. 2019. URL: `https://lists.torproject.org/pipermail/tor-dev/2019-April/013790.html` (visited on 06/14/2019).

[Kra20] Neal Krawetz. *Deanonymizing Tor Circuits*. The Hacker Factor Blog. 2020. URL: `https://www.hackerfactor.com/blog/index.php?/archives/868-Deanonymizing-Tor-Circuits.html` (visited on 02/26/2020).

[LH11] Peter Lofgren and Nicholas Hopper. “FAUST: Efficient, TTP-free Abuse Prevention by Anonymous Whitelisting”. In: *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*. WPES ’11. ACM, 2011.

[Liu+17] Zhuotao Liu et al. “TorPolice: Towards enforcing service-defined access policies for anonymous communication in the Tor network”. In: *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE Computer Society, 2017.

[Mat15] Nick Mathewson. *Denial-of-service attacks in Tor: Taxonomy and defenses*. Tech. rep. 2015-10-001. The Tor Project, 2015. URL: `https://research.torproject.org/techreports/dos-taxonomy-2015-10-29.pdf` (visited on 02/14/2020).

[Nak+08] Satoshi Nakamoto et al. *Bitcoin: A peer-to-peer electronic cash system*. 2008. URL: `https://bitcoin.org/bitcoin.pdf` (visited on 01/19/2020).

[ØS06] Lasse Øverlier and Paul Syverson. “Locating Hidden Servers”. In: *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006.

[SC18] Mehrdad Salimitari and Mainak Chatterjee. “An Overview of Blockchain and Consensus Protocols for IoT Networks”. In: *CoRR* abs/1809.05613 (2018).

[SSG99] Stuart Stubblebine, Paul Syverson, and David Goldschlag. “Unlinkable Serial Transactions: Protocols and Applications”. In: *ACM Trans. Inf. Syst. Secur.* 2.4 (1999).

[Tor19] The Tor Project. *Tor Rendezvous Specification - Version 3*. 2019. URL: `https://gitweb.torproject.org/torspec.git/tree/rend-spec-v3.txt` (visited on 09/10/2019).

[Tor20] The Tor Project. *Tor Metrics*. Website. 2020. URL: `https://metrics.torproject.org` (visited on 02/14/2020).

[Wes19] Sarah Myers West. “Data Capitalism: Redefining the Logics of Surveillance and Privacy”. In: *Business & Society* 58.1 (2019).