Onion Pass: Token-Based Denial-of-Service Protection for Tor Onion Services

Christoph Döpmann
Distributed Security Infrastructures
Technische Univerität Berlin
christoph.doepmann@tu-berlin.de

Valentin Franck
Distributed Security Infrastructures
Technische Univerität Berlin
valentin.franck@campus.tu-berlin.de

Florian Tschorsch
Distributed Security Infrastructures
Technische Univerität Berlin
florian.tschorsch@tu-berlin.de

***Abstract*—The Tor network is widely recognized as an important tool to preserve online privacy. In addition to anonymous Internet access, it allows hosting anonymous services, i. e., Onion Services. However, connecting to an Onion Service is realized in a way that makes them vulnerable to Denial-of-Service attacks (DoS). In this work, we propose Onion Pass, an extension of the Tor protocol that utilizes anonymous cryptographic tokens to mitigate the issue. Clients can solve a challenge to acquire tokens that later can be presented to the Onion Service. The Onion Service can thus differentiate between valid and malicious requests when under attack. Please note that Onion pass is agnostic on the specific challenge-response scheme and follows a design philosophy that puts Onion Services in control of the Onion Pass protocol. We implemented a prototype of Onion Pass and present experimental results that indicate its potential to prevent DoS attacks on Onion Services by reducing their CPU usage required to identify malicious requests by a factor of 47.**

***Index Terms*—Internet security, Tor network, Denial of Service**

I. Introduction

Undoubtedly, the Tor network [1] constitutes one of the most important tools for today's Internet users to protect their online privacy. For sensitive applications, Tor's *Onion Service* protocol allows users to host services anonymously. Specifically, it provides sender-receiver anonymity by concatenating two circuits that achieve sender and receiver anonymity, respectively.

The current design of Onion Services in Tor, however, is prone to denial-of-service attacks (DoS). It is easy to force an Onion Service to spend considerable amounts of computing and networking resources with only little resources required for the attacker. This attack vector has been known for several years [2], but remains an open challenge as of today.

In this paper, we present a solution to the DoS vulnerability of Tor Onion Services. Our proposed protocol, *Onion Pass*, allows clients to prove their legitimacy using cryptographic tokens. This way, Onion Services can prioritize provably valid over unverified users and thus ensure availability even when flooded with bogus requests. While the core idea to use tokens for DoS protection in Tor is not new [3], Onion Pass takes the deliberate architectural decision to have Onion Services issue access tokens for their service themselves, instead of introducing a central token-issuing authority. This does not only give the Onion Service maximum flexibility for choosing the desired level of protection and means of authentication, but also allows building upon established cryptographic solutions. To this end, Onion Pass leverages Privacy Pass [4] as its cryptographic basis.

ISBN 978-3-903176-39-3 © 2021 IFIP

We show that our approach is actually feasible and deployable by extending the Onion Service protocol in a backwards-compatible manner. In particular, we implement Onion Pass as a prototype and show in real-world experiments on the Tor network that it can help prevent DoS attacks on Onion Services by reducing the CPU usage required to identify malicious requests by a factor of 47.

The contributions of this paper are the following:

- We present the design of Onion Pass, an extension to the Tor network for effective DoS mitigation against Onion Services.
- By providing a prototypical implementation, we demonstrate the feasibility of building and deploying Onion Pass on the Tor network.
- We analyze Onion Pass's effectiveness against DoS attacks by conducting real-world measurements that indicate its improved DoS resilience.

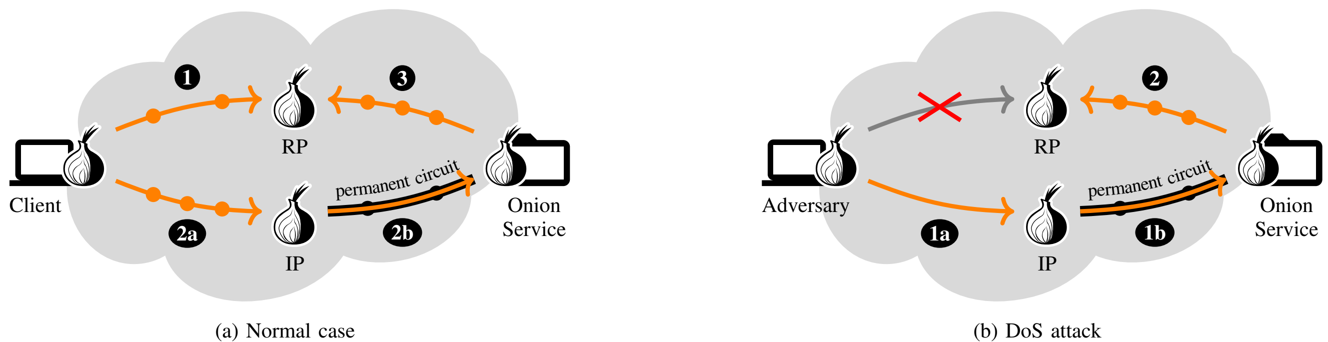
In the following, we provide a problem statement with necessary background information in Section II. In Section III, we present the technical details of our Onion Pass protocol. In Section IV, we introduce our prototypical implementation and summarize the results of our evaluation. Before concluding this work, we put our contributions into perspective of related work in Section V.

II. Problem Statement

In this section, we define the scope of our work and provide a problem statement. To this end, we first introduce the underlying attack vector's background.

A. The Tor Network

The Tor network [1] is an anonymity system based on the principle of *onion routing* [5]. It is comprised of thousands of servers operated by volunteers and organizations. Anonymity is achieved by relaying users' streams of communication over multiple intermediate nodes. Clients construct such *circuits* by choosing three random relays from the Tor network. By applying a telescope-like cryptographic scheme of multiple



(a) Normal case

(b) DoS attack

Fig. 1. The Onion Service protocol in normal operation and during a DoS attack against an Onion Service (IP = Introduction Point, RP = Rendezvouz Point).

layers of encryption, it is ensured that each relay within a circuit only knows its immediate predecessor and successor. This way, no single entity can observe both the source and the target of any stream of communication. At a lower level, communication over circuits is realized by splitting data into packets of constant size, called *cells*. The normal use case for Tor is to access servers on the "clear" Internet anonymously. To enable this, some relays opt-in to behave as so-called *exits*. When doing so, they are used as the last relay of a circuit and connect to the desired remote server outside the Tor network. Consequently, the server cannot see the original client, but only the exit relay. Therefore, this behavior creates *sender anonymity* for the client.

B. Onion Services

Apart from sender anonymity, the Tor network also offers *sender-receiver* anonymity through its mechanism of *Onion Services* (formerly known as *Hidden Services*). In particular, Onion Services allow Tor users to host services anonymously, where traffic resides entirely within the Tor network. In order to achieve anonymity for both the client and the server, each of them constructs a circuit that are concatenated. As depicted in Figure 1a, concatenating circuits requires two designated nodes in the network: The *Rendezvous Point* is the Tor relay at which the two circuits eventually meet and which is used to exchange payload data. Together, they are also known as rendezvous circuit.

As a first step, the client constructs a circuit, where the last relay serves as the Rendezvous Point. In addition, the client needs a way to communicate the Rendezvous Point's address to the Onion Service. To this end, designated *Introduction Points* exist. Similarly to the Rendezvous Point, they serve the purpose of enabling the client and the Onion Service to communicate with each other. However, the Introduction Points are chosen by the Onion Service instead of the client. The Onion Service maintains a permanent circuit to the Introduction Points, independent of specific client requests. The Introduction Point that has to be used to connect to an Onion Service is published together with its cryptographic keys as part of the Onion Service's descriptor. These descriptors are published in a network-wide distributed hash table, called *HSDir* [6]. After connecting to the Rendezvous Point (Step 1 in Figure 1a), the client signals to the Introduction Point its intent to connect to the Onion Service using an `INTRODUCE1` cell (Step 2a). This intention is then forwarded to the Onion Service as an `INTRODUCE2` cell (Step 2b). These cells include details on the Rendezvous Point and an authentication key, which is used by the Onion Service to connect (Step 3).

C. Problem Statement: DoS Attacks on Onion Services

The Onion Service protocol unfortunately gives room to attackers for carrying out denial-of-service attacks (DoS) against Onion Services. Such attacks exploit the fact that establishing a connection requires a considerable amount of resources from the Onion Service. This resource consumption can be triggered by clients with relatively little effort, resulting in asynchronous resource requirements.

In particular, on the reception of an `INTRODUCE2` cell (Step 1b in Figure 1b), the Onion Service has to carry out two steps: First, it has to build a circuit to the specified Rendezvous Point (Step 2). This requires a certain amount of networking and computational resources. Additionally, however, the `INTRODUCE2` cell contains the first half of a cryptographic handshake between the client and the Onion Service. Therefore, the Onion Service has to carry out computationally expensive asymmetric cryptography to complete the handshake.

The asynchrony between the client and the Onion Service now stems from the fact that the client can easily shortcut the effort the Onion Service has to take. To this end, there are a number of ways to minimize the adversary's effort and maximize impact. In the `INTRODUCE1` cell, the adversary can specify any relay as a Rendezvous Point, without previously connecting to it. Also, its half of the cryptographic handshake can contain bogus information. The attacker could even consider building a shorter-than-normal circuit to the Introduction Point, i. e., connect directly (Step 1a), to further reduce its resource requirements. All of these points allow a malicious client to trigger amplified resource consumption at the Onion Service, which will eventually result in the Onion Service being unavailable due to resource exhaustion if the attacker initiates enough circuits.

It is noteworthy that the anonymity properties of Tor circuits protect the attacker as well. That is, the Onion Service has no

way of differentiating the attacker from valid clients.

In essence, this attack currently makes Onion Services vulnerable to relatively weak adversaries. Due to the underlying resource amplification, attackers require only limited resources. At the same time, they behave very similarly to regular users and launch the attack only by initiating connections to the Onion Service. Unlike other attacks, the adversary's goal consists in making a specific Onion Service unavailable instead of attacking the network as a whole. With our work, we aim to reduce the effectiveness of such attacks at least for a certain set of clients.

III. Onion Pass

In the following, we present Onion Pass, our system for protecting Tor Onion Services. It is based on cryptographic tokens that are issued by the Onion Service to clients for proving their legitimacy when accessing the Onion Service in the future.

A. Design Decisions

Any DoS protection scheme for Onion Services can only be effective if it avoids or at least reduces the resource amplification that currently allows clients to attack Onion Services with relatively little effort. In the following, we briefly review design options to derive our approach.

First of all, we considered a *Proof of Rendezvous*. In such a scheme, a client would obtain a proof from the Rendezvous Point that it has indeed established a circuit. When connecting to the Onion Service, this proof would then be included into the `INTRODUCE1` and `INTRODUCE2` cells such that the Onion Service can verify that the contained Rendezvous information is valid and has required the client to spend the same amount of resources as the Onion Service. However, we see a number of issues: Firstly, the attacker could have connected to the Rendezvous Point using a shorter-than-usual circuit in order to save resources. Even more significantly though, the Rendezvous Point—which is chosen by the client—may also be malicious, operated by the same attacker.

Instead, we pursue the idea of the client proving that she is a legitimate user by presenting a cryptographic token. Again, different approaches are conceivable: Firstly, a network-wide authority could issue such tokens. Any such solution would imply that the tokens are issued by one entity (a network-wide authority), but verified by another one (e. g., the Onion Service). For this, however, a suitable cryptographic solution is not yet available [3]. Moreover, this would likely require asymmetric cryptography for token verification. The usage of asymmetric cryptography however would diminish the protection level due to the larger computational cost on the "hot path" of handling new circuits at the Onion Service. Token verification must be as efficient as possible. Otherwise, an attacker can still exhaust the Onion Service's resources by providing a large number of invalid tokens.

We therefore decided to have the Onion Service itself issue and verify tokens. This design decision addresses the outlined weaknesses. Additionally, it provides maximum flexibility for Onion Services as we believe the requirements may vary considerably, e. g., depending on the load they can handle. It allows the Onion Service to decide on its own when a client is *legitimate*, implying different types of proofs. For instance, Onion Services may want clients to identify themselves as humans by solving a CAPTCHA or by registering a user account. In other cases, a computational challenge-response puzzle, like a proof of work, may be more appropriate (e. g., to still allow a limited amount of automation). To accommodate these needs and give the Onion Service maximum flexibility and control, we decided that the Onion Service itself should be in charge of issuing tokens instead of handing this responsibility to a network-wide authority. One drawback of this approach is that tokens can only be issued to clients who have previously been able to successfully connect to the Onion Service. We argue that, for many use cases that involve regular users, e. g., news and media outlets, this is still a clear improvement in availability.

Note that all approaches also comprise common challenges, e. g., determining the rate of token issuance and thresholds for attack detection. Such challenges imply separate research questions and are out of scope of this work.

B. System Overview

The core idea of Onion Pass consists of giving clients the ability to prove their legitimacy, using a cryptographic token. If the Onion Service is under attack and cannot accommodate all clients anymore, it can decide to require such tokens to be presented. Since these can be validated (and rejected) much more efficiently than reacting to a normal `INTRODUCE2` cell, the asymmetric resource usage between client and Onion Service is reduced and the denial-of-service attack will fail to exhaust the Onion Service's resources.

As laid out before, the Onion Service issues and verifies the tokens itself. In order to bootstrap the process—allowing clients to connect to the Onion Service for obtaining tokens to redeem during later connection attempts—the Onion Service will typically only strictly require token if it is under attack (e. g., when a predefined client connection rate is surpassed).

Figure 2 shows the schematic behavior of Onion Pass, visualizing the data flows that happen during the different protocol phases. In the setup phase, the necessary key material is distributed. For issuing tokens to the client during the next phase, the client first has to connect to the Onion Service and prove its legitimacy by solving a challenge (typically on the application layer). The challenge to be solved by the client is not specified in the Onion Pass protocol and can be realized at the Onion Service's discretion. It is therefore up to the Onion Service to decide when a client should be regarded as valid. The client then requests the Onion Service to sign a blinded token and verifies that it has been issued by the publicly known key, using the previously published key material. The client can then unblind the token and redeem it, sending it to the Onion Service when it is under attack.

Implementing a token for this purpose appears to be a nontrivial cryptographic task as a number of conditions have

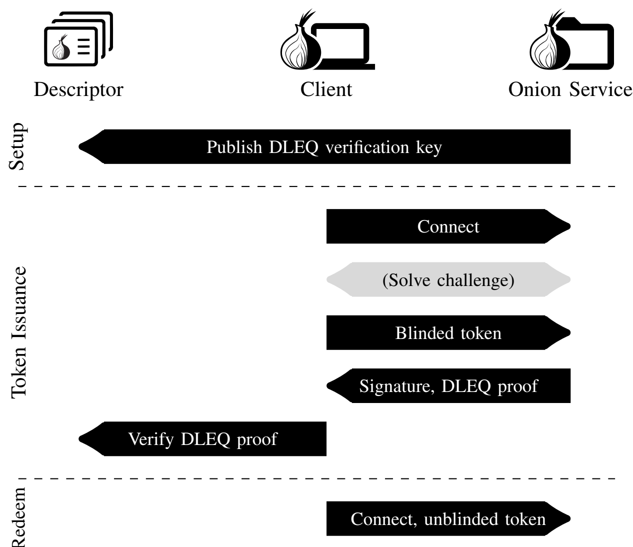


Fig. 2. Conceptual data flows in Onion Pass at the different protocol phases. The challenge/response phase is not specified by Onion Pass and is up to the Onion Service.

```
/* Setup Phase, at server */
setup() → private token signing key,
          public DLEQ verification key

/* Token Issuance, at client */
prepare_blinded_token() → blob

/* Token Issuance, at server */
sign_tokens(blobs[]) → signed_blobs[],
                       dleq_proof

/* Token Issuance, at client */
verify_dleq(dleq_proof, signed_blobs[],
            dleq_pubkey) → bool

/* Token Issuance, at client */
unblind_token(blob) → token

/* Token Redemption, at server */
verify(token) → bool
```

Fig. 3. Privacy Pass API with conceptually-named procedures derived for this paper.

to be met for application in Tor. Fortunately, however, we can build upon solid previous work that solves this task in a different context: Privacy Pass [4] introduces the cryptographic base for implementing the token in a way suitable for our use case, because of the following properties:

Firstly, it does not impact on the anonymity of the clients. In particular, after issuing the token to a user, these are unlinkable to other tokens and identities. That is, it is neither possible to determine whether the token was issued for a particular client, nor to determine whether two tokens were issued for the same client. Part of this is *key auditability*: Clients have a way to ensure that their token was generated using the same key used for any other client, such that the token is not watermarked.

Secondly, the token operations can be performed using relatively little computational effort. In particular, Onion Services are able to validate tokens with limited resource usage. If this was not the case, there might not be a benefit compared to the previous behavior that involved establishing a new circuit on every request.

And thirdly, Onion Services can prevent tokens from being used multiple times and enforce a maximum validity time. Otherwise, an attacker could simply reuse tokens or accumulate them to launch an attack.

C. Privacy Pass

Privacy Pass was introduced in [4] as a cryptographic scheme for allowing users to prove to web servers the possession of some previously issued *token*. The original use case was to avoid unnecessary CAPTCHAs when servers could not easily differentiate between users based on their IP address (e. g., because they are using Tor). This is similar to our use case because users first have to obtain an anonymous token from the server and can later use that to prove their legitimacy, without revealing their identity.

Privacy Pass can be regarded as a blind signature scheme based on elliptic curve cryptography. In essence, clients obtain tokens by having the server sign some blinded secret. Afterwards, the client unblinds the signed message, so the server will not be able to link issued tokens to those that are presented for redemption. The unblinded message can then be used by the client to prove to the server that a valid token was previously obtained. Note that, unlike RSA blind signatures, Privacy Pass does not use a key pair, but a single key.

A special feature of Privacy Pass is that it can also protect against *tagging attacks* in which a malicious server would use different keys for issuing tokens to different clients in order to be able to later differentiate between them. To protect against such attacks, Privacy Pass makes use of *discrete logarithm equivalents proofs (DLEQ)* [7]. These require that all clients share knowledge of a publicly known generator G of the server's private key which thus has to be published in advance. When issuing tokens, the server can then use DLEQs to prove to each client that the tokens were generated by the very same key, without revealing the key itself. DLEQs thus constitute non-interactive zero knowledge proofs about the key used to issue the tokens.

For the purpose of this paper, we summarize Privacy Pass's behavior into an API that allows us to easily reference the different protocol steps in the remainder of this paper. This API is depicted and commented in Figure 3.

With these features, Privacy Pass is a prime candidate for use in our scheme for DoS protection of Onion Services. In order to attribute to its core cryptographic mechanism, our system is called Onion Pass.

D. Onion Pass Details

Onion Pass denotes the integration of Privacy Pass into the Tor protocol for protecting Onion Services from denial-

of-service attacks. The extensibility of the different parts of the Tor protocol allows this embedding to be realized in a backward-compatible manner. Onion Pass needs to implement the following protocol steps:

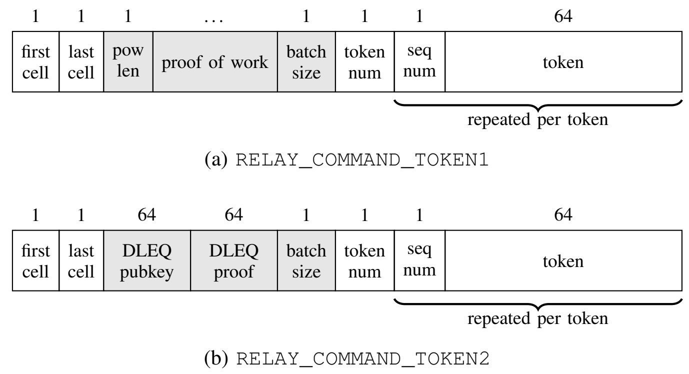
- global distribution of cryptographic key material (public key for the DLEQ proofs)
- token issuance (from the Onion Service to the client)
- token redemption (from the client to the Onion Service)

To tackle these challenges, the central approach is to extend existing Tor cell types and add new ones, as well as extend the Onion Service descriptor stored in the HSDir. Onion Pass is activated and configured by defining a set of parameters in the Tor config file (`torrc`).

a) Setup Phase/Key Distribution: In order to start using Onion Pass, an Onion Service executes Privacy Pass's `setup()` routine for generating the private signing key as well as the public DLEQ verification key. It then starts including the necessary information into its service descriptor. This step serves several purposes: Firstly, by setting an appropriate flag, the Onion Service signals to the client that it is using Onion Pass at all. As a consequence, the client can decide to include previously obtained tokens into the circuit construction process to prove its legitimacy. If the client does not yet have tokens, it can later request a number of tokens once a regular circuit has been set up to the Onion Service. Secondly, the Onion Service includes cryptographic information into its service descriptor that enables the client to carry out the DLEQ verification so the client can be certain that it has not been issued tokens from a tagged key. During token issuance, the Onion Service then includes a zero-knowledge proof indicating that the issued tokens stem from the private key represented by the public DLEQ verification key. Due to the fact that the service descriptor is accessed anonymously by the clients and cryptographically secured, this prevents Onion Services from tagging tokens by using different keys. Apart from publishing this information to the descriptor, during the setup phase, the Onion Service locally initializes a persistent storage to keep track of used tokens and avoid "double spending".

b) Token Issuance: Obtaining tokens which can later be redeemed requires the client to have a working circuit to the Onion Service at some point. Once this is established, the client can request tokens to be issued by the Onion Service. For this purpose, Onion Pass adds a new Tor cell type called `RELAY_COMMAND_TOKEN1`. We further refer to it simply as `TOKEN1`, but the full name indicates that this is an end-to-end command between the client and the Onion Service. Figure 4a displays the structure of this cell.

The main component of `TOKEN1` cells is a sequence of blinded tokens to be signed. More specifically, these are generated by the conceptual `prepare_blinded_token()` function of Privacy Pass. The client can request the issuance of many tokens at once. However, Tor cells have a fixed size. The full sequence of tokens may be distributed across multiple cells, which is indicated using the `first_cell` and `last_cell` flags. Moreover, the first cell contains the total number of tokens to sign.



(a) `RELAY_COMMAND_TOKEN1`

(b) `RELAY_COMMAND_TOKEN2`

Fig. 4. Structure of the new cell types introduced by Onion Pass. The gray parts only appear in the first cell of each sequence.

When the full series of `TOKEN1` cells was received by the Onion Service, the Onion Service checks the client's legitimacy. This step assures that tokens are not issued to automated bots, for example. Since requirements vary for each Onion Service, we intentionally leave this step unspecified. Onion Pass, however, offers two general approaches to prove a client's legitimacy: Firstly, the client can include the response to a challenge in its `TOKEN1` cells (the first `TOKEN1` cell contains a variable-size field for this purpose). This mechanism may be used if the client's legitimacy can be proven in an automated way, for example. Likely, the more common way is to check the client's legitimacy on the application layer. For instance, the client may be requested to solve a CAPTCHA when using the Onion Service's website. Likewise, the client might be required to authenticate. In either way, the Onion Service application server signals to the Tor daemon whether the check was successful by issuing an appropriate command via the Tor daemon's control port.

Upon successful verification of the request, the Onion Service then makes use of the `sign_tokens()` procedure, providing it the previously received blinded tokens. This step generates the signed tokens as well as the DLEQ proof for the whole batch of tokens. Both pieces of information are then returned to the client using `RELAY_COMMAND_TOKEN2` cells, another novel type of cell shown in Figure 4b.

The reason why it is recommended to request and issue complete batches of tokens instead of single tokens at a time is that the computational effort for generating the DLEQ proof is mostly independent of the number of tokens it encompasses. Therefore, larger batch sizes can significantly reduce the resource usage of issuing tokens for the Onion Service.

Once the client has received signed tokens as well as the DLEQ proof via `TOKEN2` cells, it first verifies that the tokens were signed with the publicly auditable key known from the Onion Service's descriptor (instead of a custom, tagged one). Therefore, the client uses the `verify_dleq()` procedure. Afterwards, the client unblinds each issued token, using `unblind_token()`, so the Onion Service cannot match them to previously issued tokens. At this point, the tokens are ready for future use.

c) Token Redemption: Previously issued tokens can later be used to prove the client's legitimacy when the Onion Service is under a DoS attack. To this end, we make use of the extensibility of Tor's cell specification [8]. In particular, we extend the `INTRODUCE1` and `INTRODUCE2` cells to optionally carry a token to be redeemed. This is possible because these cell types are already defined by the Tor specification to allow optional extension headers.

The client includes an (unblinded) token in the `INTRODUCE1` cell, which is forwarded by the Introduction Point to the Onion Service as part of an `INTRODUCE2` cell. Once this request to establish a rendezvous circuit arrives at the Onion Service, the Onion Service can use this token to decide on the legitimacy of the client. Using the `verify()` procedure, the Onion Service can tell whether the unblinded token stems from a blinded token the Onion Service had issued before. If it is under attack (defined by a threshold in the Tor config), it may choose to reject any requests that either have an invalid token or lack a token at all. If the Onion Service is not under attack, it will typically also allow requests such requests in order to enable the clients to proof their legitimacy and obtain tokens.

Due to the fact that verifying the token takes much less computational and networking resources than establishing a circuit to the (possibly faked) Rendezvous Point, this constitutes the desired DoS protection. Once the token has been used, the Onion Service adds it to its local storage of used tokens. This way, it can reject tokens that are used more than once. In the simplest case, this may be a hash set, but in order to retain a constant size of the data structure over time, an appropriately-dimensioned Bloom filter may also be used [4]. Note that, in the case of Bloom filters, false positives may occur (tokens are wrongly rejected). However, this will not put the Onion Service to danger because it cannot happen that tokens are wrongly accepted multiple times. If a token is not accepted, the client can use another one.

d) Key Rotation: From time to time, the Onion Service will need to rotate its keys. Otherwise, clients could, over time, still accumulate large amounts of tokens for launching a coordinated attack. The exact point in time to rotate keys (both private and public DLEQ key) is not fixed and can be triggered at the Onion Service's discretion. The protocol allows multiple keys to co-exist for a certain time to enable a graceful transition, if desired by the Onion Service. For this, clients include the public DLEQ key during redemption, so the Onion Service decide which key to verify against. Eventually, however, key rotation will invalidate previously issued tokens. Keys should therefore not be rotated too often if infrequent users of the Onion Service should also retain the ability to access the Onion Service when it is under attack. The Onion Service may decide to either define a fixed interval of time or a number of issued tokens after which keys are rotated.

IV. Evaluation

In this section, we present the results of our initial evaluation of the effectiveness of Onion Pass as a DoS protection scheme for Onion Services. In this regard, the effectiveness of Onion Pass depends on two general factors: the security properties of the underlying cryptographic scheme and the achieved CPU load reduction for the Onion Service. We will investigate those two aspects separately after giving some details on the implementation of our prototype.

A. Implementation

We implemented the Onion Pass protocol in the original Tor software, branching off version 0.4.1. The result is a prototype usable for evaluation of the overall approach, in which non-central system aspects like the actual user challenges and key rotation have not been implemented yet. Onion Pass is currently only implemented for the newer version 3 Onion Services. In total, we authored 5,707 lines of code in 48 source code files (another 3,582 lines of code were generated automatically by Tor's `trunnel` framework for cell parsing).

Since the publicly available implementation of Privacy Pass is written in Go, our implementation includes our own version, implemented in the C language. Since Tor's cryptography API was too limited for this, we based the cryptography on `openssl`. In particular, we used the `secp256v1` elliptic curve with a bit length of 256 per curve point coordinate. Moreover, we employed SHA-256 as the main hash function from which we derived, e. g., the necessary message authentication codes (MAC). For hashing random numbers to points on the elliptic curve, we implemented the *Simplified Shallue-Woestijne-Ulas* algorithm (Simplified SWU) [9] as proposed in Privacy Pass. As token replay prevention, we used a simple hash set for storing the used tokens.

All in all, the prototype allows us to further evaluate Onion Pass's performance and shows that it is feasible to be implemented in the Tor code base. We make available the implementation as an open source project[1].

B. Security Properties

Due to its strong reliance on Privacy Pass many of Privacy Pass's security properties are directly inherited by Onion Pass assuming that the underlying elliptic curves cryptography is secure. In particular, Onion Pass provides strong *misauthentication resistance*, that is, the cryptographic implementation of Privacy Pass ensures that only tokens issued by the Onion Service will be regarded as valid. Moreover, *limited replayability* is achieved in Onion Pass by remembering used tokens at the Onion Service. Also, tokens cannot be accumulated arbitrarily over an infinite amount of time. Instead, they have *limited validity*, depending on the validity of the Onion Service's token signing key, which is rotated regularly.

Especially in the context of Tor, it is crucial not to harm the clients' anonymity. Again, Onion Pass ensures this by relying on the security properties of Privacy Pass: Tokens are *unlinkable*, also due to the *key auditability* of the system. Clients use the DLEQ proofs sent with the issued tokens and the public key from the service descriptor to verify that they

[1] `https://github.com/cdoepmann/onion-pass`

TABLE I
CPU TIME FOR RENDEZVOUS CIRCUIT CONSTRUCTION VS. TOKEN VALIDATION TIME IN ONION PASS.

Operation	**Mean time** $[\mu s]$	**Stddev** $[\mu s]$
Rendezvous circuit construction	6,416	1,019
Token validation	134	5
Ratio	47.8	—

TABLE II
CPU TIME FOR CRYPTOGRAPHIC OPERATIONS.

Entity	**Operation**	**Mean time** $[\mu s]$	
		per batch	per token
Client	Token generation, blinding	3,123	104
	Signature unblinding and DLEQ verification	4,156	139
	Token redemption	—	8
Onion Service	Token signature (incl. DLEQ)	3,482	116
	Token verification	—	114

have not been tagged. In addition, the token blinding that happens within Privacy Pass realizes *backward anonymity*: Issued tokens cannot later be assigned to users because the blinding factors used during issuance are ephemeral.

The effectiveness of Onion Pass for mitigating the described denial-of-service vulnerability primarily depends on the performance of validating tokens. If invalid tokens can be identified as such with considerably less computational effort than establishing a circuit to the Rendezvous Point, the asymmetric resource requirements between the attacker and the Onion Service are effectively avoided. We therefore focus on evaluating the performance of token validation. However, care has to be taken not to introduce a new denial-of-service attack surface. We informally argue that this is indeed not the case. The reason for this is that both, token validation but also token issuance, are efficient operations. Moreover, an attacker cannot easily stop the Onion Service from issuing tokens to benign users when they have at least once connected to the Onion Service, because there is no strict upper limit on the number of issued tokens. Accumulating large amounts of tokens on the other side can be prevented by the Onion Service by making the application-layer challenge hard enough. For the same reason, creating Sybils does not benefit the attacker. We note again, the limitation of Onion Pass is that it only succeeds in ensuring the availability of the Onion Service for users who have previously accessed it, obtaining at least one token. Then, however, clients can utilize this connection for obtaining more tokens that ensure being able to access the Onion Service continuously in the future.

Please note that our prototype implementation does not constitute a production-ready software version. For this, a more thorough security analysis should be conducted.

C. Performance

The performance of Onion Pass is a crucial factor determining its effectiveness as a DoS protection. Recall that denial-of-service attacks against Onion Services work by making the Onion Service spend large amounts of CPU power on constructing circuits to Rendezvous Points only on the basis of bogus client requests. Onion Pass allows Onion Services to save this effort by validating client tokens beforehand. Therefore, it can only be effective if validating tokens requires clearly less work than constructing a circuit.

In order to validate this property, we used our prototype implementation of Onion Pass. We set up both an Onion Service and a client running our modified version on the Tor network. The client repeatedly established circuits to the Onion Service while we measured resource usage at the server side. We chose this evaluation strategy due to several reasons: Primarily, it allows us to investigate the performance of Onion Pass as a real-world implementation in the original Tor code base. This is important because benchmark results are required for comparing against vanilla Tor. Therefore, alternative popular evaluation approaches like event-discrete simulations are not applicable. Evaluating the effectiveness of denial-of-service protections is difficult, because the available resources cannot easily be modeled or replicated. We therefore chose to stick to the real Tor network as closely as possible. Please note that conducting our measurements safely on the Tor network is possible without putting users or the network itself to danger. Since we use a dedicated Onion Service under our control and we do not flood the network with connection requests, our measurements do not impact Tor users or the network.

While measuring CPU time for token validation is straightforward, determining the CPU effort for circuit construction required to instrument many different parts of the Tor software to avoid capturing delays that do not stem from computing, but from networking or waiting. We therefore manually identified the relevant pieces of code and added time measurement procedures, yielding microsecond resolution. We repeated the measurement 1,000 times. The experiments were carried out on a desktop computer with an AMD Ryzen 3 1200 quad-core processor and 16 GB of RAM. Table I shows the results. We can see that token validation is indeed extremely fast. In fact, it takes more than 47 times less CPU power than constructing the Rendezvous circuit on average. As a consequence, attackers that do not have a sufficient amount of tokens (which the Onion Service can control during issuance) can cause dramatically less damage to an Onion Service. Note that the standard deviation of the Rendezvous circuit construction time is relatively high because a large portion of that time is spent during circuit selection, which is a random process that also influences its runtime. The times presented in Table I also include computational time needed for parsing/processing the cells.

Apart from token redemption, we also verify that both, server and client, can handle the additional effort for generating, issuing and redeeming tokens. For this, we focus on the raw CPU time used for only the respective cryptographic operations. Again, we took these measurements from 1,000 accesses to the Onion Service. Moreover, we defined a batch

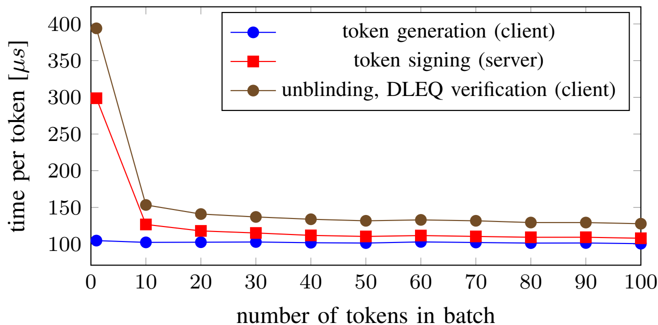


Fig. 5. CPU time for issuing token batches of different size.

size of 30 tokens during issuance. The results are depicted in Table II. We can see that the required CPU work does not constitute an obstacle towards real-world adoption since the required computational effort per token, as well as for complete batches, is very small, at the order of little more than 100 μs per token.

As outlined before, it is recommended to issue complete batches of tokens instead of single tokens at a time because this can reduce the overhead needed for the computation of the DLEQ proofs. We quantified this relationship by measuring the time needed to issue differently-sized batches of tokens and calculated the per-token computation time. Figure 5 displays the result. As can be seen, the per-token time for both, generating the token signatures and DLEQ proofs, as well as for verifying the DLEQ proofs is clearly reduced with batches of more than one token. This is due to the fact that the DLEQ computation and verification takes almost constant time, independently of the batch size. Consequently, it is desirable to issue multiple tokens at once. However, the performance gain when surpassing a batch size of approximately 50 is negligible. From this point onwards, the time per token does not further decrease and potential further benefits are masked by jitter.

V. Related Work

Due to the general increase in DoS attacks on the public Internet and against the Tor network in particular, previous research has analyzed those attacks and presented various potential mitigations for them.

In [2], Nick Mathewson from the Tor Project gives an overview over different classes of DoS attacks against Tor and discusses possible mitigations for them. In [10], Borisov et al. analyze the impact that some DoS attacks can have on anonymity networks, including the Tor network. The authors find that DoS attacks are not only a threat to availability, but may also help deanonymize Tor users. This attack vector was used in [11] to force user to select attacker-controlled circuits. The authors of [12] and [13] measure the effectiveness of DoS attacks in Tor and propose two potential detection algorithms. The feasibility of DoS attacks on Tor was illustrated in [14]. In contrast to previous papers, our work focuses on preventing attacks that specifically target Onion Services—not clients that use Tor to access the regular Internet anonymously.

Bocovich et al. [15] were the first to describe DoS attacks that exploit the Onion Service design for resource amplification. It was based on observations from the first large DoS attack against the Tor network targeting Onion Services in December 2017. In 2019, the weakness was addressed and a mitigation was implemented in Tor [16]. It allows Onion Services to have their Introduction Point enforce a rate limiting for `INTRODUCE1` cells. This way, the availability of the network is protected by limiting rendezvous circuit constructions. The general DoS vulnerability of Onion Services, however, remains. Moreover, it does not protect the availability of the Onion Service itself. In contrast, Onion Pass makes Onion Services available even if under attack, focusing not only on the network but on the service as well.

One approach to achieve this would be to require users to spend a scarce resource, such as computational power. Therefore, proof of work may be a conceivable option [17]. Previous work has indeed considered to require solving cryptographic puzzles for DoS protection in Tor [18]. The approach, however, did not focus on Onion Services. While cryptographic proof of work might be viable for certain non-interactive applications, it comes with other issues such as adequately tuning the difficulty for users with devices of differing computational power. Consequently, Onion Pass also allows to integrate proof of work schemes (and even reserves suitable fields in its cell structures), but is generally agnostic about the kind of challenge the Onion Service would like to implement.

Cryptographic tokens have been proposed for Tor before, e. g. as incentives for contributing resources [19]–[21]. However, they can also be used for DoS mitigation strategies, such as Onion Pass, that can be regarded as authentication or allow list schemes [22]. The fundamental idea is to give some clients precedence over others that may constitute attackers, making resource exhaustion more difficult. This direction has also been investigated before: Faust by Lofgren and Hopper [23] is an anonymous whitelisting scheme without the need for a trusted third party. It relies on client signatures and unlinkable serial transactions [24]. However, it requires a long-term user identity and is not tailored to Tor and Onion Services in particular. Onion Pass instead relies on Privacy Pass [4]. Privacy Pass satisfies the requirements for anonymous whitelisting and does not require a long-term user identity. It was intended to be applied in a different use case (reducing CAPTCHA requests for users accessing websites over heavily shared Internet access, such as Tor). Our work presenting Onion Pass, however, shows that it can serve as a building block for protecting Onion Services from DoS attacks as well.

VI. Conclusion

In this work, we presented Onion Pass, a protocol extension for the Tor network protecting Onion Services from DoS attacks. Onion Pass works by giving precedence to clients who can prove their legitimacy using previously obtained cryptographic tokens. Our prototypical implementation shows that Onion Pass can clearly increase the robustness of Onion Services against resource exhaustion attacks.

Acknowledgments

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG—German Research Foundation, TS 477/1-1, project 451732324).

References

[1] R. Dingledine, N. Mathewson, and P. F. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[2] N. Mathewson, "Denial-of-service attacks in Tor: Taxonomy and defenses," The Tor Project, Tech. Rep. 2015-10-001, 2015. [Online]. Available: https://research.torproject.org/techreports/dos-taxonomy-2015-10-29.pdf

[3] G. Kadianakis, "How to stop the onion denial (of service)," 2020. [Online]. Available: https://blog.torproject.org/stop-the-onion-denial

[4] A. Davidson *et al.*, "Privacy pass: Bypassing internet challenges anonymously," *Proceedings on Privacy Enhancing Technologies*, 2018.

[5] P. F. Syverson, D. M. Goldschlag, and M. G. Reed, "Anonymous connections and onion routing," in *Proceedings of the 18th IEEE Symposium on Security and Privacy*, 1997.

[6] The Tor Project, "Tor directory protocol specification," 2020. [Online]. Available: https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt

[7] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *Advances in Cryptology*. Springer, 1993.

[8] T. Tor Project, "Tor rendezvous specification - version 3," 2020. [Online]. Available: https://gitweb.torproject.org/torspec.git/tree/rend-spec-v3.txt

[9] A. Faz-Hernández *et al.*, "Hashing to elliptic curves," Internet Draft, Work in Progress, 2019. [Online]. Available: https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-05/

[10] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz, "Denial of service or denial of security?" in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[11] R. Jansen, F. Tschorsch, A. Johnson, and B. Scheuermann, "The sniper attack: Anonymously deanonymizing and disabling the Tor network," in *Proceedings of the NDSS Symposium*, 2014.

[12] N. Danner, S. Defabbia-Kane, D. Krizanc, and M. Liberatore, "Effectiveness and detection of denial-of-service attacks in tor," *ACM Transactions on Information and System Security*, 2012.

[13] N. Danner, D. Krizanc, and M. Liberatore, "Detecting denial of service attacks in tor," in *Financial Cryptography and Data Security*, 2009.

[14] R. Jansen, T. Vaidya, and M. Sherr, "Point break: A study of bandwidth denial-of-service attacks against tor," in *Proceeding of the 28th USENIX Security Symposium*, 2019.

[15] C. Bocovich *et al.*, "Addressing denial of service attacks on free and open communication on the internet," Tech. Rep. 2019-05-001, 2019. [Online]. Available: https://research.torproject.org/techreports/dos-censorship-report2-2019-05-31.pdf

[16] D. Goulet and G. Kadianakis, "Proposal 305: Establish_intro cell dos defense extension," Tor Developer Mailing List, 2019. [Online]. Available: https://github.com/torproject/torspec/blob/master/proposals/305-establish-intro-dos-defense-extention.txt

[17] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Proceedings of the 12th Annual International Cryptology Conference*, ser. Lecture Notes in Computer Science. Springer, 1992.

[18] N. Fraser, D. Kelly, R. Raines, R. Baldwin, and B. Mullins, "Using client puzzles to mitigate distributed denial of service attacks in the tor anonymous routing environment," in *Proceedings of the IEEE International Conference on Communications*, 2007.

[19] T. Ngan, R. Dingledine, and D. S. Wallach, "Building incentives into tor," in *Proceedings of the 14th International Conference on Financial Cryptography and Data Security*, 2010.

[20] R. Jansen, N. Hopper, and Y. Kim, "Recruiting new tor relays with BRAIDS," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[21] R. Jansen, A. Miller, P. Syverson, and B. Ford, "From onions to shallots: Rewarding tor relays with TEARS," in *Proceedings of the 14th Privacy Enhancing Technologies Symposium*, 2014.

[22] R. Henry and I. Goldberg, "Formalizing anonymous blacklisting systems," in *2011 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2011.

[23] P. Lofgren and N. Hopper, "Faust: Efficient, ttp-free abuse prevention by anonymous whitelisting," in *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*. ACM, 2011.

[24] S. Stubblebine, P. Syverson, and D. Goldschlag, "Unlinkable serial transactions: Protocols and applications," *ACM Transactions on Information and System Security*, 1999.