ONIONS:
TOR-POWERED DISTRIBUTED DNS
FOR ANONYMOUS SERVERS

by

Jesse Victors

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Ming Li Major Professor	Dr. Nicholas Flann Committee Member
Dr. Daniel Watson Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2015

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Onion Routing

As the prevalence of the Internet and other communication has grown, so too has the development and usage of privacy-enhancing systems. These are tools and protocols that provide privacy by obfuscating the link between a user's identification or location and their communications. Privacy is not achieved in traditional Internet connections because SSL/TLS encryption cannot hide IP and TCP headers, which must be exposed to allow routing between two parties; eavesdroppers can easily break user privacy by monitoring these headers. A closely related property is anonymity – a part of privacy where user activities cannot be tracked and their communications are indistinguishable from others. Tools that provide these systems hold a user's identity in confidence, and privacy and anonymity are often provided together. Following a general distrust of unsecured Internet communications and in light of the 2013-current revelations by Edward Snowden of Internet mass-surveillance by the NSA, GCHQ, and other members of the Five Eyes, users have increasingly turned to these tools for their own protection. Privacy-enhancing and anonymity tools may also be used by the military, researchers working in sensitive topics, journalists, law enforcement running tip lines, activists and whistleblowers, or individuals in countries with Internet censorship. These users may turn to proxies or VPNs, but these tools often track their users for liability reasons and thus rarely provide anonymity. Furthermore, they can easily voluntarily or be forced to break confidence to destroy user privacy. More complex tools are needed for a stronger guarantee of privacy and anonymity.

Today, most anonymity tools descend from mixnets, an early anonymity system invented by David Chaum in 1981. [1] In a mixnet, user messages are transmitted to one

or more mixes, who each partially decrypt, scramble, delay, and retransmit the messages to other mixes or to the final destination. This enhances privacy by heavily obscuring the correlation between the origin, destination, and contents of the messages. Mixnets have inspired the development of many varied mixnet-like protocols and have generated significant literature within the field of network security. [2] [3]

Mixnet descendants can generally be classified into two distinct categories: high-latency and low-latency systems. High-latency networks typically delay traffic packets and are notable for their greater resistance to global adversaries who monitor communication entering and exiting the network. However, high-latency networks, due to their slow speed, are typically not suitable for common Internet activities such as web browsing, instant messaging, or the prompt transmission of email. Low-latency networks, by contrast, do not delay packets and are thus more suited for these activities, but they are more vulnerable to timing attacks from global adversaries. [4] In this work, we detail and introduce new functionality within low-latency protocols.

Onion routing is a technique for enhancing privacy of TCP-based communication across a network and is the most popular low-latency descendant of mixnets in use today. It was first designed by the U.S. Naval Research Laboratory in 1997 for military applications [5] [6] but has since seen widespread usage. In onion routing with public key infrastructure (PKI), a user selects a set network nodes, typically called *onion routers* and together a *circuit*, and encrypts the message with the public key of each router. Each encryption layer contains the next destination for the message – the last layer contains the message's final destination. As the *cell* containing the message travels through the network, each of these onion routers in turn decrypt their encryption layer like an onion, exposing their share of the routing information. The final recipient receives the message from the last router, but is never exposed to the message's source. [3] The sender therefore has privacy because the recipient does not know the sender's location, and the sender has anonymity if no identifiable or distinguishing information is included in their message.

The first generation of onion routing used circuits fixed to a length of five, assumed a

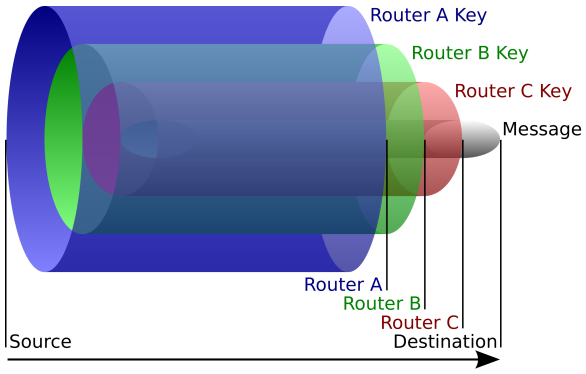

Figure 1.1: An example cell and message encryption in an onion routing scheme. Each router "peals" off its respective layer of encryption; the final router exposes the final destination.

static network topology, and most notably, introduced the ability to mate two circuits at a common node or server. This last capability enabled broader anonymity where the circuit users were anonymous to each other and to the common server, a capability that was adopted and refined by later generation onion routers. Second generation introduced variable-length circuits, multiplexing of all user traffic over circuits, exit policies for the final router, and assumed a dynamic network by routing updates throughout the network. A client, Alice, in second-generation onion routers also distributed symmetric keys through the cell layers. If routers remember the destinations for each message they received, the recipient Bob can send his reply backwards through the circuit and each router re-encrypts the reply with their symmetric key. Alice unwraps all the layers, exposing the Bob's reply. The transition from public-key cryptography to symmetric-key encryption significantly reduced the CPU load on onion routers and enabled them to transfer more packets in the same amount of time. However, while influential, first and second generation onion routing networks have fallen out of use in favor of third-generation systems. [3]

1.2 Tor

Tor is a third-generation onion routing system. It was invented in 2002 by Roger Dingledine, Nick Mathewson, and Paul Syverson of the Free Haven Project and the U.S. Naval Research Laboratory [4] and is the most popular onion router in use today. Tor inherited many of the concepts pioneered by earlier onion routers and implemented several

key changes: [3] [4]

- **Perfect forward secrecy:** Rather than distributing keys via onion layers, Tor clients negotiate ephemeral symmetric encryption keys with each of the routers in turn, extending the circuit one node at a time. These keys are then purged when the circuit is torn down; this achieves perfect forward secrecy, a property that ensures that the encryption keys will not be revealed if long-term public keys are later compromised.

- **Circuit isolation:** Second-generation onion routers mixed cells from different circuits in realtime, but later research could not justify this as an effective defence against an active adversary. [3] Tor abandoned this in favor of isolating circuits from each other inside the network. Tor circuits are used for up to 10 minutes or whenever the user chooses to rotate to a fresh circuit.

- **Three-hop circuits:** Previous onion routers used long circuits to provide heavy traffic mixing. Tor removed mixing and fell back to using short circuits of minimal length. With three relays involved in each circuit, the first node (the *guard*) is exposed to the user's IP address. The middle router passes onion cells between the guard and the final router (the *exit*) and its encryption layer exposes it to neither the user's IP nor its traffic. The exit processes user traffic, but is unaware of the origin of the requests. While the choice of middle and exits can be routers can be safely random, the guard nodes must be chosen once and then consistently used to avoid a large cumulative chance of leaking the user's IP to an attacker. This is of particular importance for circuits from hidden services. [7] [8]

- **Standardized to SOCKS proxy:** Tor simplified the multiplexing pipeline by transitioning from application-level proxies (HTTP, FTP, email, etc) to a TCP-level SOCKS proxy, which multiplexed user traffic and DNS requests through the onion circuit regardless of any higher protocol. The disadvantage to this approach is that Tor's client software has less capability to cache data and strip identifiable information out of a protocol. The countermeasure was the Tor Browser, a fork of Mozilla's open-source

Firefox with a focus on security and privacy. To reduce the risks of users breaking their privacy through Javascript, it ships with the NoScript extension which blocks all web scripts not explicitly whitelisted. The browser also forces all web traffic, including DNS requests, through the Tor SOCKS proxy, provides a Windows-Firefox user agent regardless of the native platform, and includes many additional security and privacy enhancements not included in native Firefox. The browser also utilizes the EFF's HTTPS Everywhere extension to re-write HTTP web requests into HTTPS whenever possible; when this happens the Tor cell contains an additional inner encryption layer.

- **Directory servers:** Tor introduced a set of trusted directory servers to distribute network information and the public keys of onion routers. Onion routers mirror the digitally signed network information from the directories, distributing the load. This simplified approach is more flexible and scales faster than the previous flooding approach, but relies on the trust of central directory authorities. Tor ensures that each directory is independently maintained in multiple locations and jurisdictions, reducing the likelihood of an attacker compromising all of them. [3] We describe the contents and format of the network information published by the directories in section 1.2.3.

- **Dynamic rendezvous with hidden services:** In previous onion routers, circuits mated at a fixed common node and did not use perfect forward secrecy. Tor uses a distributed hashtable to record the location of the introduction node for a given hidden service. Following the initial handshake, the server and the client then meet at a different onion router chosen by the client. This approach significantly increased the reliability of hidden services and distributed the communication load across multiple rendezvous points. [4] We provide additional details on the hidden service protocol in section 1.2.4 and our motivation for addition infrastructure in section 1.3.

As of March 2015, Tor has 2.3 million daily users that together generate 65 Gbit/s of traffic. Tor's network consists of nine authority nodes and 6,600 onion routers in 83 countries. [9] In a 2012 Top Secret U.S. National Security Agency presentation leaked by

Edward Snowden, Tor was recognized as the "the king of high secure, low latency Internet anonymity". [10] [11] In 2014, BusinessWeek claimed that Tor was "perhaps the most effective means of defeating the online surveillance efforts of intelligence agencies around the world." [12]

1.2.1 Design

Tor's design focuses on being easily deployable, flexible, and well-understood. Tor also places emphasis on usability in order to attract more users; more user activity translates to an increased difficulty of isolating and breaking the privacy of any single individual. Tor however does not manipulate any application-level protocols nor does it make any attempt to defend against global attackers. Instead, its threat model assumes that the capabilities of adversaries are limited to observing fractions of Tor traffic, that they can actively delay, delete, or manipulate traffic, that they may attempt to digitally fingerprint packets, that they may run onion routers themselves, and that they may compromise a fraction of other existing routers. Together, most of the assumptions may be broadly classified as traffic analysis attacks. Tor's final focus is defending against these types of attacks. [4]

1.2.2 Circuit Construction

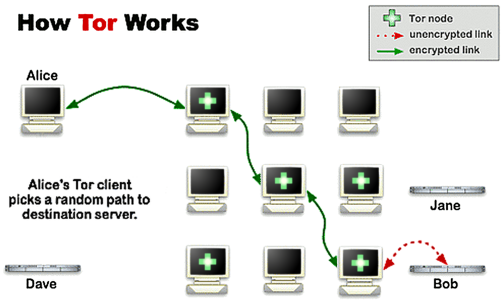

Figure 1.2: A circuit through the Tor network.

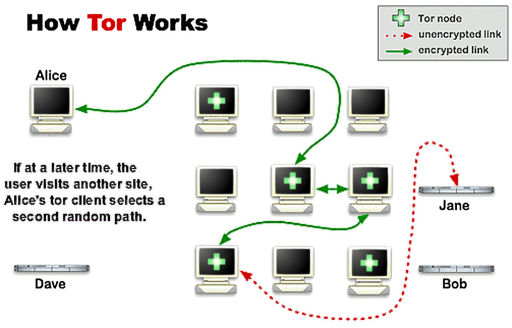

Figure 1.3: A Tor circuit is changed periodically, creating a new user identity.

Overview

Let Alice be a client who wishes to enhance her privacy by using Tor but does not run a Tor router herself. Her basic procedure for initializing her own circuit is as follows:

1. Alice selects a Tor router R_1 and sets up an authenticated encrypted connection to it.
2. Alice selects a second router R_2 and tells R_1 to build a TCP link to it.
3. Alice uses her R_1 tunnel to establish an authenticated encrypted connection with R_2.
4. Alice selects an R_3 and uses the $Alice \leftrightarrow R_1 \leftrightarrow R_2$ tunnel to tell R_2 to connect to R_3.
5. Alice establishes an authenticates encrypted connection with R_3 over the $Alice \leftrightarrow R_1 \leftrightarrow R_2$ tunnel.

Following the successful circuit construction, she can then send messages out of R_3 through the $Alice \leftrightarrow R_1 \leftrightarrow R_2 \leftrightarrow R_3$ tunnel.

As a defense against traffic analysis using packet size, Alice packs and pads circuit traffic into equally-sized Tor *cells* of 512 bytes and changes R_2 and R_3 every 10 minutes. [13]

todo: Explain why Alice must choose her guard node carefully and then stick with it.

TAP

The Tor Authentication Protocol (TAP) is a critical piece of Tor networking infrastructure as it allows Alice to verify the authenticity of the onion routers that she selects for her circuit while still remaining anonymous herself. Namely it prevents active spoofing attacks, which if successful would deanonymize Alice and compromise her privacy.

First, TAP assumes that the following is established:

- Alice has a reliable PKI that can securely distribute the identity, IP addresses, and public keys of all Tor routers. Tor accomplishes this through consensus documents from authority nodes, described in section 1.2.3.
- Let $\mathcal{E}_B$ mean encryption and $\mathcal{D}_B$ mean decryption under B's public-private keypair for some party B.

- p is a prime such that $q = \frac{q-1}{2}$ is also prime and let g be a generator of the subgroup of $\mathbb{Z}_p^*$ of order q.
- Let R_L be a generator that returns uniformly random L-bit values in the interval $[1, \min(q, 2^L) - 1]$.
- Define f as SHA-1 which takes input from $\mathbb{Z}_p$ and returns a L_f-bit value.

Todo: I'd like to include that TAP diagram [14] [4] and find out about the forward-backwards keys.

Then TAP proceeds as:

1. (a) Alice selects a Tor router, R_i.
   (b) Alice generates $x = R_L$ and computes $s = g^x \bmod p$.
   (c) Alice sends $c = \mathcal{E}_{R_i}(s)$ to R_i.
2. (a) R_i computes $m = \mathcal{D}_{R_i}(c)$ and confirms that $1 < m < p - 1$.
   (b) R_i generates $y = R_L$ and computes $a = g^y \bmod p$ and $b = f(m^y \bmod p)$.
   (c) R_i sends (a, b) to Alice.
3. Alice confirms that $1 < a < p - 1$ and that $b = f(a^x \bmod p)$.
4. If the assertions pass, then Alice has confirmed R_i's identity and now Alice and R_i can use $a^x = m^y$ as a shared symmetric encryption key and encrypt messages under AES.
5. Alice repeats steps 1 – 4 until the circuit is of the desired length.

Thus for each router in the circuit, Alice perform one public-key encryption and her half of a Diffie-Hellman-Merkle (DHE) handshake, and each router in turn performs one private-key decryption and their half of a DHE handshake. Under the assumptions that RSA is one way, AES remains unbroken, and f is strong and acts as a random oracle, an attacker has only a negligible chance of being able to impersonate R_i and read messages that she tunnels through the circuit. [15]

NTor

In mid 2013, TAP was superseded by "NTor", a new protocol invented by Goldberg, Stebila, and Ustaoglu. [16] Compared to TAP, NTor replaced the computational cost associated with DHE exponentiation with significantly more efficient elliptic curve cryptography (ECC). Its implementation uses Curve25519, [17] a Montgomery curve designed by Bernstein to be used for high-speed ECDHE key exchanges. All Curve25519 operations are implemented to run in $\mathcal{O}(1)$ time.

NTor is initialized by defining the following,

- Let $H(x, k)$ be HMAC-SHA256, which accepts a message x and yield a 32-byte MAC output. Let k_1, k_2, and k_3 be some fixed secret keys for the HMAC algorithm, $k_1 \neq k_2 \neq k_3$.
- Let $C_{gen}()$ generate a Curve25519 keypair. This function requires 32 bytes from a cryptographically secure source (such as a CSPRNG) to generate the private key, then it derives the public key.
- Let $C_{sec}(pvt, pub)$ yield a 32-byte common secret given a Curve25519 private and public key, *pvt* and *pub*, respectively.
- Let each Tor router R_j generate $b_j, B_j = C_{gen}()$ and publish B_j through the consensus document.
- Let *id* by the router's identification fingerprint.

Then NTor proceeds as:

1. (a) Alice selects a Tor router, R_i.
   (b) Alice generates $x, X = C_{gen}()$ and sends X to R_i.
2. (a) R_i generates $y, Y = C_{gen}()$.
   (b) R_i sets $sec = C_{sec}(y, X) \parallel C_{sec}(b, X) \parallel id \parallel X \parallel Y$.
   (c) R_i computes $seed = H(sec, k_1)$.

(d) R_i computes $auth = H(H(sec, k_2) \parallel id \parallel B \parallel Y \parallel X, k_3)$.

(e) R_i sends Y and $auth$ to Alice.

(f) R_i deletes the y, Y keypair.

3. (a) Alice sets $sec = C_{sec}(x, Y) \parallel C_{sec}(x, B) \parallel id \parallel X \parallel Y$.

   (b) Alice computes $seed = H(sec, k_1)$.

   (c) Alice confirms that $auth = H(H(sec, k_2) \parallel id \parallel B \parallel Y \parallel X, k_3)$.

4. Alice and R_i now have a shared secret *seed* that is then used indirectly for symmetric key encryption.

1.2.3 Consensus Documents

Todo: this section is incomplete. Here I plan to detail the consensus documents that I will be using, since I use them in my Solution section.

The Tor network is maintained by nine authority nodes, who each vote on the status of nodes and together hourly publish a digitally signed consensus document containing IPs, ports, public keys, latest status, and capabilities of all nodes in the network. The document is then redistributed by other Tor nodes to clients, enabling access to the network. The document also allows clients to authenticate Tor nodes when constructing circuits, as well as allowing Tor nodes to authenticate one another. Since all parties have prior knowledge of the public keys of the authority nodes, the consensus document cannot be forged or modified without disrupting the digital signature. [18]

cached-certs

This section is incomplete. See https://collector.torproject.org/formats.html

cached-microdescs

This section is incomplete. See Tor's dir-spec.txt, section 3.3

cached-microdesc-consensus

This section is incomplete. See https://collector.torproject.org/formats.html

1.2.4 Hidden Services

Although Tor's primary and most popular use is for secure access to the traditional Internet, Tor also supports *hidden services* – anonymous servers hosting services such as websites, marketplaces, or chatrooms. These servers intentionally mask their IP addresses through Tor circuits and thus cannot normally be accessed outside the context of Tor. In contrast to Tor-anonymized web requests where the client is anonymous but the server is known, Tor hidden services provide bidirectional anonymity where both parties remain anonymous and never directly communicate with one another. [19]

Tor does not contain a DNS system for its websites; instead hidden service addresses are algorithmically generated by generating the SHA-1 hash of their public RSA key, truncating to 80 bits, and converting the remainder to base58. Ignoring the possibility of SHA-1 collisions, this builds a one-to-one relationship between a hidden service's public key and its address which can be confirmed without requiring any identifiable information or central authorities. The address is the appended with the .onion Top-Level Domain (TLD).

A hidden server, Bob, first builds Tor circuits to several random relays and enables them to act as *introduction points* by giving them its public key, B_K. He then then uploads his public key and the fingerprint identity of these nodes to a distributed hashtable inside the Tor network, signing the result. He then publishes his hidden service address in a backchannel. When Alice obtains this address and enters it into a Tor-enabled browser, her software queries this hashtable, obtains B_K and Bob's introduction points, and builds a Tor circuit to one of them, ip_1. Simultaneously, the client also builds a circuit to another relay, rp, which she enables as a rendezvous point by telling it a one-time secret, sec. During this procedure, hidden services must continue to use their same entry node in order to avoid leaking its IP address to possibly malicious onion routers. [7] [8]

She then sends to ip_1 a cookie encrypted with B_K, containing rp and sec. Bob decrypts this message, builds a circuit to rp, and tells it sec, enabling Alice and Bob to communicate.

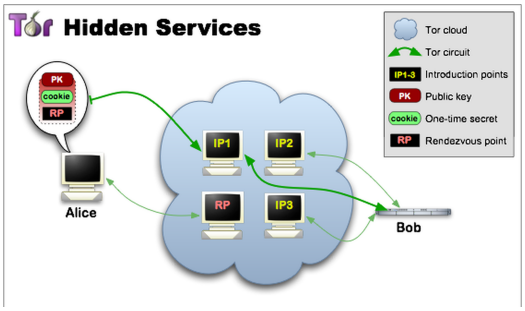

Figure 1.4: Alice uses the encrypted cookie to tell Bob to switch to *rp*.

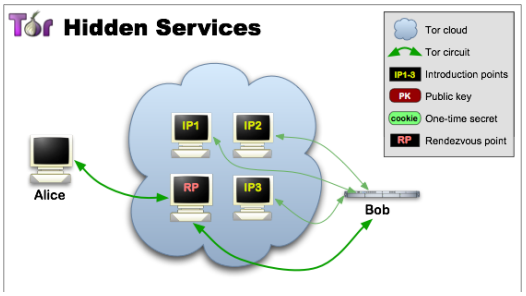

Figure 1.5: Bidirectional communication between Alice and the hidden service.

Their communication travels through six Tor nodes: three established by Alice and three by Bob, so both parties remain anonymous. From there traditional HTTP, FTP, SSH, or other protocols can be multiplexed over this new channel.

As of March 2015, Tor hosts approximately 25,000 unique hidden services that together generate around 450 Mbit/s of traffic. [9]

1.3 Motivation

As Tor's hidden service addresses are algorithmically derived from the service's public RSA key, there is at best limited capacity to select a human-meaningful name. Some hidden service operators have attempted to work around this issue by finding an RSA key by brute-force that generates a partially-desirable hidden service address (e.g. "example0uyw6wgve.onion") and although some alternative encoding schemes have been proposed, (section) the problem generally remains. The usability of hidden services is severely challenged by their non-intuitive and unmemorable base58-encoded domain name. For example, 3g2upl4pq6kufc4m.onion is the address for the DuckDuckGo search engine, suw74isz7wqzpmgu.onion is a WikiLeaks mirror, and 33y6fjyhs3phzfjj.onion and vbmwh445kf3fs2v4.onion are both SecureDrop instances for anonymous document submission. These addresses maintain strong privacy as the strong association between its public key and its address significantly breaks the association between the service's purpose and its address. However, this privacy comes at a cost: it is impossible to classify or label a hidden service's purpose in advance, a fact well known within Tor hidden service communities.

Over time, third-party directories – both on the Clear and Dark Internets – appeared in attempt to counteract this issue, but these directories must be constantly maintained and furthermore this approach is not convenient nor does it scale well. This suggests the strong need for a more complete and reliable solution.

1.4 Contributions

Our contribution to this problem is six-fold:

- We introduce a novel distributed naming system that enables Tor hidden service operators to choose a unique human-meaningful domain name and construct a strong association between that domain name and their hidden service address, squaring Zooko's Triangle (section **??**).
- We described a new distributed, publicly confirmable, and partially self-healing transactional database.
- We rely on the existing Tor network and other infrastructure, rather than introduce a new network. This simplifies our assumptions and reduces our threat model to attack vectors already known and well-understood on the Tor network.
- We introduce a novel protocol for proving the non-existence of any domain name.
- We enable Tor clients to verify the authenticity of a domain name against its corresponding hidden service address with minimal data transfers and without requiring any additional queries.
- We preserve the privacy of both the hidden service and the anonymity of Tor clients connecting to it.

CHAPTER 2

SOLUTION

2.1 Overview

We propose the Onion Name Service, or OnioNS, a system which uses enables transparent translation of .tor domain names to .onion addresses. The system has three main aspects: the generation of self-signed claims on domain names by hidden service operators, the processing of domain information within the OnioNS servers, and the receiving and authentication of domain names by a Tor client.

First, a hidden service operator, Bob, generates an association claim between a meaningful domain name and a .onion address. Without loss of generality, let this be example.tor → example0uyw6wgve.onion. For security reasons we do not introduce a central repository and authority from which Bob can purchase domain names; however, domains are not trivially obtainable and Bob must expend effort to claim and maintain ownership of example.tor. We achieve this through a proof-of-work scheme. Proof-of-work systems are noteworthy for their asymmetry: they require the issuer to spend effort to find an answer to a moderately hard computational problem, but once solved can be easily verified correct by any recipient. The requirement of proof-of-work fulfils three main purposes:

1. Significantly reduces the threat of record flooding.

2. Introduces a barrier-of-entry that encourages the utilization of domain names and the availability of the underlying hidden services.

3. Increases the difficulty of domain squatting, a denial-of-service attack where a third-party claims one or more valuable domain names for the purpose of denying or selling

them en masse to others. In Tor's anonymous environment, this vector is particularly prone to Sybil attacks.

Bob then digitally signs his association and the proof-of-work with his service's private key.

Second, Bob uses a Tor circuit to anonymously transmit his association, proof-of-work, digital signature, and public key to OnioNS nodes, a subset of Tor routers. This set is deterministically derived from the volatile Tor consensus documents and is rotated periodically. The OnioNS nodes receive Bob's information, distribute it amongst themselves, and later archive it in a sequential transaction database, of which each OnioNS node holds their own copy. The nodes also digitally sign this database and distribute their signatures to the other nodes. Thus the OnioNS Tor nodes maintain a common database and have each vouched for its authenticity.

Third, a Tor client, Alice, uses a Tor client to anonymously connect to one of these OnioNS nodes and request a domain name. Without loss of generality, let this request be example.tor. Alice receives Bob's "example.tor → example0uyw6wgve.onion" association, his proof-of-work, his digital signature, and his public key. Alice then verifies the proof-of-work and Bob's signature against his public key, and hashes his key to confirm the accuracy of example0uyw6wgve.onion. Alice then looks up example0uyw6wgve.onion in Tor's distributed hashtable, finds Bob's introduction point, and confirms that its knowledge of Bob's public key matches the key she received from the OnioNS nodes. Finally, she finishes the Tor hidden service protocol and begins communication with Bob. In this way, Alice can contact Bob through his chosen domain name without resorting to use lower-level hidden service addresses. The uniqueness and authenticity of the Bob's domain name is maintained by the subset of Tor nodes.

2.2 Definitions

To discuss OnioNS precisely we must first define some central terms that we will use throughout the rest of this document. We detail their exact contents in section 2.5 and how

they are used throughout sections 2.3 and 2.6.

domain name

A *domain name* is a case-insensitive identification string claimed by a hidden service operator. The syntax of OnioNS domain names mirrors the Clearnet DNS; we use a sequence of name-delimiter pairs with the .tor Top Level Domain (TLD). The TLD is a name at depth one and is preceded by names at sequentially increasing depth. The term "domain name" refers to the identiciation string as a whole, while "second-level domain" refers to the central name that is immediately followed by the TLD, as illustrated in Figure 2.1. Domain names point to *destinations* – other domain names with either the .tor or .onion TLD.

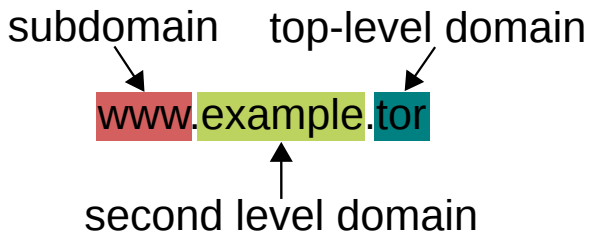

Figure 2.1: A sample domain name: a sequence of labels separated by delimiters. In OnioNS, hidden service operators build associations between second-level domain names and their hidden service address.

The Clearnet DNS defines a hierarchy of administrative realms that are closely tied to the depth of each name. By contrast, OnioNS makes no such distinction; we let hidden service operators claim second-level names and then control all names of greater depth under that second-level name. Hidden service operators may then choose to administer or sell the use of their subdomains, but this is outside the scope of this project.

Record

A *Record* is a small textual data structure that contains one or more domain-destination pairs, a proof-of-work, a digital signature, and a public key. Records are issued by hidden service operators and sent to OnioNS servers. Every Record is self-signed with the hidden service's key. In section 2.5.1 we describe the five different types of Records:

Create, Modify, Move, Renew, and Delete. A Create Record represents a registration on an unclaimed second-level domain name, Modify, Move, and Renew are operations on that domain name or its subdomains, and a Delete Record relinquishes ownership over the second-level domain name and all its subdomains.

broadcast

The term for the protocol described in section 2.6.1 that describes the uploading of new Records to OnioNS servers through Tor circuits and then the subsequent confirmation that the Record has been received.

Snapshot

A *Snapshot* is textual database designed to hold collections of Records in a short-term volatile cache. They are also used to transmit sets of Records between OnioNS servers.

Page

A *Page* is textual database designed to archive one or more Records in long-term storage. Pages are held and digitally signed by OnioNS nodes and are writable only for fixed periods of time before they are read-only. Each Page contains a link to a previous Page, forming an append-only public ledger known as an *Pagechain*. This forms a two dimensional distributed data structure: the chain of Pages grows over time and there are multiple redundant copies of each Page spread out across the network at any given time.

Mirror

A *Mirror* is any machine (inside or outside of the Tor network) that has performed a synchronization (section **??**) against the OnioNS network and now holds a complete copy of the Pagechain. Mirrors do not actively participate in the OnioNS network and do not have the power to manipulate the central page-chain.

Quorum Candidate

A *Quorum Candidates* are *Mirrors* inside the Tor network that have also fulfilled

two additional requirements: 1) they must demonstrate that they are an up-to-date Mirror, and 2) that they have sufficient CPU and bandwidth capabilities to handle powering OnioNS in addition to their regular Tor duties. In other words, they are qualified and capable to power OnioNS, but have not yet been chosen to do so.

Quorum

A *Quorum* is a subset of Quorum Candidates who have active responsibility over maintaining the master OnioNS Pagechain. Each Quorum node actively its own Page, which has a lifetime of that Quorum. The Quorum is randomly chosen from Quorum Candidates as described in section 2.6.3.

flood

The term for the periodic burst of communication that occurs between Quorum nodes wherein Snapshots are exchanged and each Quorum node obtains the state and digital signature of the current Page maintained by all other Quorum nodes. This communication is described in section 2.6.2.

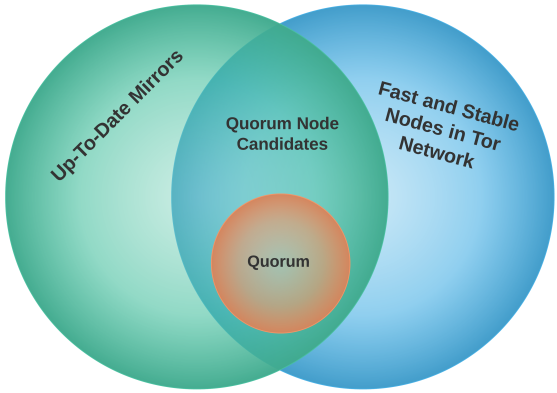

Figure 2.2: The relationship between Mirrors, Quorum Candidates, and the Quorum. A Mirror is any machine that holds the OnioNS Pagechain, Quorum Candidates are both up-to-date Mirrors and reliable Tor nodes, and the Quorum is randomly selected from the pool of Quorum Candidates.

Throughout the rest of this document, let Alice be a Tor client, and Bob be the operator of a hidden service. Assume that neither Alice nor Bob run Tor relays themselves, but let

them only use Tor. Therefore Alice and Bob know the public keys of the Tor authority nodes and they can obtain and fully verify any past or current set of consensus documents. Bob has access to his hidden service private RSA key and may or may not have a PGP key with a subkey for email encryption.

2.3 Basic Design

OnioNS is build on a fundamental sequence of operations, as illustrated in Figure 2.3:

1. Bob generates a valid Record.
2. Bob broadcasts the Record to the Quorum.
3. The Record is flooded across the Quorum.
4. Each Quorum node stores the Record into its current Page at the head of its local Pagechain.
5. Alice queries the system for a domain name.
6. Alice receives back the Record matching that domain name and confirms its validity.
7. Alice connects to Bob via the hidden service protocol.

These steps describe the propagation of any Record throughout the entire network. The protocols for each party are described in sections 2.6.1, 2.6.2, and 2.6.3, respectively. It should be noted that the activities of Alice, Bob, and the Quorum occur asynchronously: hidden service operator may be transmitting Records to the Quorum at the same time that clients are querying for other domain names.

2.3.1 Domain Name Operations

Bob first generates a valid Record which he transmits over a Tor circuit to a randomly-selected Quorum node, who confirms acceptance or returns an error message otherwise. Bob's Record is flooded to the rest of Quorum via the periodic flushing of Snapshots by

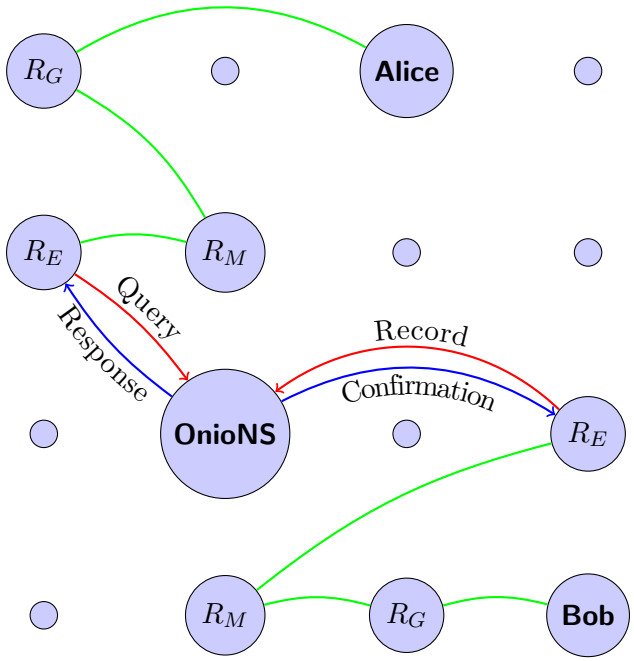

Figure 2.3: Bob uses a Tor circuit (guard, middle, and exit Tor routers) to anonymously broadcast a record to OnioNS. Alice uses her own Tor circuit to query the system for a domain name, and she is given Bob's record in response. Then Alice connects to Bob by Tor's hidden service protocol (section 1.2.4).

each Quorum node. Therefore, Bob can confirm that the rest of the Quorum received his Record by constructing another circuit to a different randomly-selected Quorum node and asking it for his Record. This process is illustrated in Figure 2.4.

2.3.2 Pagechain Maintenance

The Pagechain is OnioNS' fundamental data structure and is the central component of the entire system. It is a distributed append-only transactional database of a fixed maximum length. Each Page references a previous Page, forming a long chain. The head of the chain (the latest Page) is maintained and digitally signed by each member of the current Quorum. The Pagechain is designed as a public and fully confirmable data structure – that is, any Mirror can check the integrity of each Page and the Records contained within them, the uniqueness of domain names, and the validity of the digital signatures on each Page from all Quorum nodes.

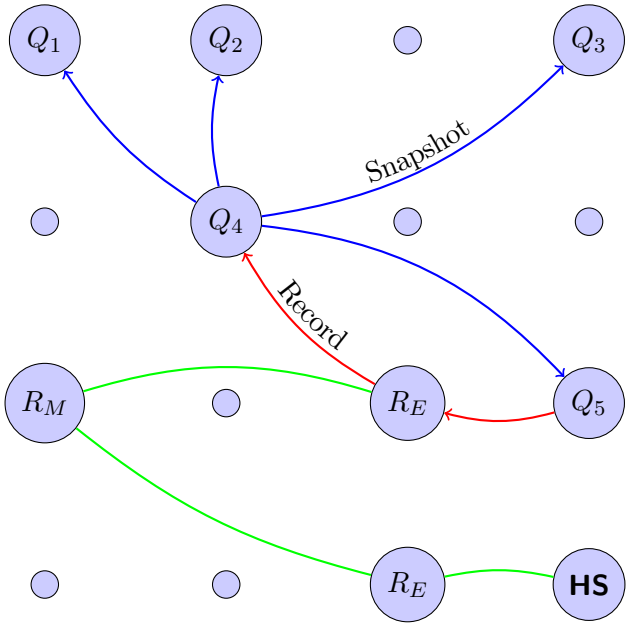

Figure 2.4: Bob uses his existing circuit (green) to inform *quorum* node Q_4 of the new record. Q_4 then floods it via *Snapshots* to all other *Quorum* nodes. Each node stores it in their own *Page* for long-term storage. Bob confirms from another *Quorum* node Q_5 that his Record has been received.

Any Mirror may continue its own Pagechain by generating a new Page, adding that Page to the front of its Pagechain, deleting the oldest Page, and then merging new Records into the new Pagechain head. When a Quorum Candidate is chosen as a Quorum node, it modifies its local Pagechain in this way. Assuming that the entire Quorum is honest and maintains perfect flooding communication, the entire Quorum would be in agreement. However, any Quorum node may experience downtime and may miss Snapshot broadcasts, have data corruption, act maliciously, or have other cause for its Pagechain head to deviate relative to the others. Therefore, disagreements may inevitably form within the Quorum. We address this under our original assumption that the largest set of Quorum nodes that have matching and valid Pagechains are also acting honestly. Therefore each Mirror can follow this rule to derive the master Pagechain and safely ignore any "sidechains". Likewise, each new honest Quorum member follows suite when creating a new Page, as described in section 2.6.2. Thus the Pagechain is at least partially self-healing. We illustrate this in Figure 2.5.

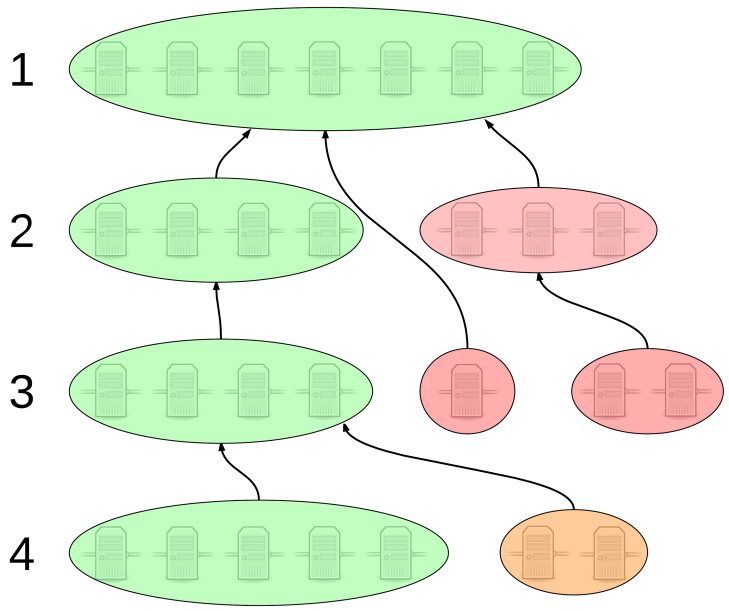

Figure 2.5: An example Pagechain across four Quorums. Each Page contains a references to a previous Page, forming a distributed append-only chain. Quorum_1 is honest and maintains reliable flooding communication, and thus has identical Pages. Quorum_2's largest cluster is likewise in agreement, but there are three nodes which are acting maliciously together and have changed their Pages. Node 5 in Quorum_3 references an old page in attempt to bypass Quorum_2's records, and nodes 6-7 are colluding with nodes 5-7 from Quorum_2. Finally, Quorum_4 has two nodes that acted honestly but did not record new records, so their Pagechains differ from the others. However, across all four days the largest clusters are honest nodes and thus integrity remains in the master Pagechain.

2.3.3 Client Request

Let Bob and Charlie be two hidden services, and assume that OnioNS knows a Record from Bob containing an "example.tor → example0uyw6wgve.onion" association, and a Record from Charlie containing "example2.tor → example5chqt7to6.onion, sub.example2.tor → example.tor". Now let Alice request "sub.example2.tor".

Since the .tor TLD is not used by the Clearnet DNS, Alice's client software must direct the request through a Tor circuit to some OnioNS resolver, O_r. By default, O_r belongs to the set of Quorum Candidate nodes although Alice may override this and choose some Mirror if she wishes. For security and load reasons, Quorum nodes refuse to respond to queries, so Alice cannot ask them. Following Alice's request, O_r searches its local Pagechain for "example2.tor" and finds Charlie's Record, which it then sends to Alice. Alice sees the

"sub.example2.tor → example.tor" association and now queries O_r for "example.tor". O_r then locates and sends Bob's Record to Alice. Alice sees that "example.tor" has a destination of "example0uyw6wgve.onion", and because it has a .onion address she does not need to issue any more queries. Instead, Alice connects to Bob's example0uyw6wgve.onion service via the Tor hidden service protocol and sends her original request ("sub.example2.tor") to Bob. As is the case with the Clearnet, Bob's web server may or may not provide specific content to Alice based on this request, but his configuration is outside our scope.

In this way, Alice can query an OnioNS resolver for some Charlie-selected domain name and recursively resolve it to a .onion address transparently. The number of resolutions Alice makes is fixed to a maximal length (section 2.5.1) so this algorithm always completes. We discuss minor optimizations and security enhancements to this procedure in section 2.6.3.

2.4 Primitives

2.4.1 Cryptographic

OnionNS makes use of cryptographic hash algorithms, digital signatures, proof-of-work, and a pseudorandom number generator.

- Let $H(x)$ be a cryptographic hash function. In our reference implementation we define $H(x)$ as SHA-384, a truncated and slight modificated derivative of SHA-512. Currently the best preimage attack of SHA-512 breaks 57 out of 80 rounds in 2^{511} time. [20] SHA-384 is included in the National Security Agency's Suite B Cryptography for protecting information classified up to Top Secret.

- Let $S_d(m, r)$ be a deterministic RSA digital signature function that accepts a message m and a private RSA key r and returns an RSA digital signature. Let $S_d(m, r)$ use $H(x)$ as a digest function on m in all use cases. In our reference implementation we define $S_d(m, r)$ as EMSA-PSS, (EMSA4) a probabilistic signature scheme defined by PKCS1 v2.1 and republished in 2003's RFC 3447.

- Let $S_{ed}(m, c)$ be an Ed25519 digital signature function that accepts a message m and a private Curve25519 key c and returns a 64-byte Ed25519 digital signature. Ed25519 is digital signature scheme designed with high performance, strong implementation security, and minimal keypair and signature sizes objectives in mind. The Ed25519 elliptic curve is birationally equivalent to Curve25519, and Curve25519 keys can be converted to Ed25519 keys in $\mathcal{O}(1)$ time. [21] Let $S_{ed}(m, c)$ use $H(x)$ as a digest function on m in all use cases.

- Let $V_{ed}(m, C)$ validate an Ed25519 digital signature by accepting a message m and a public Curve25519 key C, and return true if the signature is valid. Let $V_{ed}(m, C)$ use $H(x)$ as a digest function on m in all use cases.

- Let PoW(i) be a proof-of-work scheme such as a strong key derivation function that accepts an input key k and returns a derived key. Our reference implementation uses a fixed salt and scrypt, a password-based key derivation function which is notable for its large memory and CPU requirements during its operation. The scrypt function provides significantly greater resistance to custom hardware attacks and massively parallel computation primarily due to its memory requirements. This limits attackers to the same software implementation and asymptotic cost as legitimate users. [22] [23] We choose scrypt because of these advantages over other key derivation functions such as SHA-256 or PBKDF2. For these reasons scrypt is also common for proof-of-work purposes in some cryptocurrencies such as Litecoin.

- Let $R(s)$ be a pseudorandom number generator that accepts an initial seed s and returns a list of numerical pseudorandom numbers. We suggest MT19937, commonly known as the Mersenne Twister. This generator is widely used throughout most programming languages and is well known for its speed, long period, and the high quality of its pseudorandom output, although it is not cryptographically secure. [24]

2.4.2 Symbols

- Let L_Q represent size of the Quorum.
- Let L_T represent the number of routers in the Tor network.
- Let L_P represent the maximum number of Pages in the Pagechain.
- Let q be an Quorum iteration counter.
- Let Δq be the lifetime of a Quorum in days: every Δq days q is incremented by one and a new Quorum is chosen.
- Let s be a Snapshot iteration counter.
- Let Δs be the lifetime of a Snapshot in days: every Δs minutes s is incremented by one, a flood happens, and a fresh Snapshot is used.

All textual databases are encoded in JSON. JSON is significantly more compact than XML, but retains readability. Its support of basic primitive types is highly applicable to our needs. Additionally, we consider the JSON format safer than byte-level encoding.

2.5 Data Structures

2.5.1 Record

There are five different types of Records: Create, Modify, Move, Renew, and Delete. The latter four Records mimic the format of the Create Record with minor exceptions.

Create

A Create record consists of nine components: *type*, *nameList*, *contact*, *timestamp*, *consensusHash*, *nonce*, *pow*, *recordSig*, and *pubHSKey*. Fields that are optional are blank unless specified, and all fields are encoded in base64, except for *nameList* and *timestamp*, which are encoded in standard UTF-8.

Field	**Required?**	**Description**
type	Yes	A textual label containing the type of Record. In this case, *type* is set to "Create".
nameList	Yes	An array list of domain names and their destinations. The list of domain names includes one or more second-level domain name, while the remainder of the list contains zero or more subdomains (names at level < 2) under that second-level domains. In this way, records can be referenced by their unique second-level domain names. Destinations use either .tor or .onion TLDs.
contact	No	Bob's PGP key fingerprint, if he has a key and chooses to disclose it. If the fingerprint is listed, clients may query a public keyserver for this fingerprint, obtain the PGP public key, and contact Bob over encrypted email.
timestamp	Yes	The UNIX timestamp of when the issuer created the registration and began PoW(i).
consensusHash	Yes	The hash of the consensus document that generated $Quorum_q$
nonce	Yes	Four bytes that serve as a source of randomness for PoW(i).
pow	Yes	16 bytes that store the output of PoW(i).
recordSig	Yes	The output of $S_d(m, r)$ where $m = nameList \| timestamp \| consensusHash \| nonce \| pow$ and r is the hidden service's private RSA key.

pubHSKey	Yes	Bob's public RSA key.

Table 2.1: A Create record, which contains fields common to all records. Every record is self-signed and must have verifiable proof-of-work before it is considered valid.

Modify

A Modify record allows an owner to update his registration with updated information. The Modify record has identical fields to Create, but *type* is set to "Modify". The owner corrects the fields, updates *timestamp* and *consensusHash*, revalidates the proof-of-work, and transmits the record. Modify records have a difficulty of $\frac{\text{difficulty}_{\text{Create}}}{4}$. Modify records can be used to add and remove domain names but cannot be used to claim additional second-level domains.

Move

A Move record is used to transfer one or more second-level domain names and all associated subdomains from one owner to another. Move records have all the fields of a Create record, have their *type* is set to "Move", and contain two additional fields: *target*, a list of domain-destination pairs, and *destPubKey*, the public key of the new owner. Domain names and their destinations contained in *target* cannot be modified; they must match the latest Create, Renew, or Modify record that defined them. Move records also have a difficulty of $\frac{\text{difficulty}_{\text{Create}}}{4}$.

Renew

Second-level domain names (and their associated domain names) expire every L days because (as explained below) the page-chain has a maximum length of L pages. Renew records must be reissued periodically at least every L days to ensure continued ownership of domain names. Renew records are identical to Create records, except that *type* is set to

"Renew". No modifications to existing domain names can be made in Renew records, and the domain names contained within must already exist in the page-chain. Similar to the Modify and Move records, Renew records have a difficulty of $\frac{\text{difficulty}_{\text{Create}}}{4}$.

Delete

If a owner's private key is compromised or if they wish to relinquish ownership rights over all of their domain names, they can issue a Delete record. Aside from their *type* field set to "Delete", Delete records are identical in form to Create records but have the opposite effect: all second-level domains contained within the record are purged from local caches and made available for others to claim. There is no difficulty associated with Delete records, so they can be issued instantly.

2.5.2 Snapshot

Snapshots contain four fields: *originTime*, *recentRecords*, *fingerprint*, and *snapshotSig*.

originTime

Unix time when the snapshot was first created.

recentRecords

A array list of records in reverse chronological order by receive time.

fingerprint

The hash of the public key of the machine maintaining this snapshot. This is the Tor fingerprint, a unique identification string widely used throughout Tor infrastructure and third-party tools to refer to specific Tor nodes.

snapshotSig

The output of $S_{ed}(originTime \,\|\, recentRecords \,\|\, fingerprint], c)$ where c is the machine's private Curve25519 NTor key.

2.5.3 Page

Each page contains five fields, *prevHash*, *recordList*, *consensusDocHash*, *fingerprint*, and *pageSig*.

prevHash

The SHA-384 hash of *prevHash*, *recordList*, and *consensusDocHash* of a previous page.

recordList

An array list of records, sorted in a deterministic manner.

consensusDocHash

The SHA-384 of *cd*.

fingerprint

The hash of the public key of the machine maintaining this page, the Tor fingerprint.

pageSig

The output of $S_{ed}(H(prevHash \parallel recordList \parallel consensusDocHash)], c)$ where c is the machine's private Curve25519 NTor key.

2.6 Protocols

Throughout this section, when a party downloads consensus documents, they can safely obtain them from any source because they can validate the signatures on the documents against the Tor authority public keys to ensure that they were not modified in transit. Several online sources archive consensus documents and make them available retrospectively; in our reference implementation we also introduce another source. Note that in practice it is often more efficient to download consensus documents en-masse as they achieve very high compression ratios under 7zip.

2.6.1 Hidden Services

Record Generation

Mirrors and Tor clients check the validity of Records that they receive through synchronization or query responses, respectively. Invalid Records will be rejected by all parties, so Bob must generate a valid Record before broadcast.

1. Bob selects the value for *type* based on the desired operation.
2. Bob constructs the *nameList* domain-destination associations.
3. Bob provides his PGP key fingerprint in *contact* or leaves it blank if he doesn't have a PGP key or if he chooses not to disclose it. Bob can derive his PGP fingerprint with the "gpg –fingerprint" Unix command.
4. Bob records the number of seconds since the Unix epoch in *timestamp*.
5. Bob sets *consensusHash* to the output of $H(x)$, where x is the consensus document published at 00:00 GMT on day $\lfloor \frac{q}{\Delta q} \rfloor$.
6. Bob initially defines *nonce* as four zeros.
7. Let *central* be *type* $\|$ *nameList* $\|$ *contact* $\|$ *timestamp* $\|$ *consensusHash* $\|$ *nonce*.
8. Bob sets *pow* as PoW(*central*).
9. Bob sets *recordSig* as the output of $S_d(m, r)$ where $m = central \| pow$ and r is Bob's private RSA key.
10. Bob saves the PKCS.1 DER encoding of his RSA public key in *pubHSKey*.

Bob then must increment *nonce* and reset *pow* and *recordSig* until $H(central \| pow \| recordSig) \leq 2^{difficulty * count}$ where *difficulty* is a fixed constant that specifies the work difficulty and *count* is the number of second-level domain names claimed in the Record. An example of a completed and valid record is shown in Figure 2.6.

```
0  {
1      "names": {
2          "example.tor": "exampleruyw6wgve.onion"
3      }
4      "contact": "AD97364FC20BEC80",
5      "timestamp": 1424045024,
6      "consensusHash": "uU0nuZNNPgilLlLX2n2r+sSE7+N6U4DukIj3rOLvzek=",
7      "nonce": "AAAABw==",
8      "pow": "4I4dzaBwi4AIZW8s2m0hQQ==",
9      "recordSig":      "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/
           kBEpyXRSopzjoFs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh+
           QOnKl0fKBN7fqowjkQ3ktFkR0Vuox9WrrbNTMa4+
           up0Np52hlbKA3zSRz4fbR9NVlh6uuQ=",
10     "pubHSKey": "MIGhMA0GCSqGSIb3DQEBAQUAA4GPADCBiwKBgQDE7CP/
           kgwtJhTTc4JpuPkvA7Ln9wgc+
           fgTKgkyUp1zusxgUAn1c1MGx4YhO42KPB7dyZOf3pcRk94XsYFY1ULkF2+
           tf9KdNe7GFzJyMFCQENnUcVXbcwLH4vAeiGK7R/nScbCbyc9LT+
           VE1fbKchTL1QzLVBLqJTxhR+9YPi8x+QIFAdZ8BJs="
11 }
```

Figure 2.6: A sample registration record. The textual fields are in UTF-8, while the binary fields are in base64. The structure is encoded in JSON.

Record Validation

Let Carol be a Tor client or a Mirror that receives a Record from another Mirror. Bob must also perform these procedures to ensure that he sends a valid Record.

1. Carol checks that the Record contains valid JSON.

2. Carol checks that *type* is either “Create”, “Modify”, “Move”, “Renew”, or “Delete”.

3. Carol checks that *nameList* is has a length $\in [1, 24]$, contains at least one second-level domain name, and that all subdomains have a second-level domain name within *nameList*. Additionally, Carol checks that there is no domain name nor destination that uses more than 16 names, that is longer than 32 characters, or whose total length is more than 128 characters.

4. Carol checks that *contact* is either 0, 16, 24, or 32 characters in length, and that *contact* is valid hexadecimal.

5. Carol checks that *timestamp* is not in the future nor more than 48 hours old relative to her system clock.

6. Carol checks that *pubHSKey* is a valid PKCS.1 DER-encoded public 1024-bit RSA key, and that when converted to a .onion address that address appears at least once in *nameList*.

7. Carol checks the validity of *recordSig* deterministic signature against *pubHSKey* and $central \parallel pow$.

8. Carol obtains the $\lfloor \frac{q}{\Delta q} \rfloor$ 00:00 GMT consensus document and checks $H(x)$ on it against *consensusHash*. She also confirms that the consensus document is no older than $\min(48, 12 * \Delta q)$.

9. Carol checks that $H(central \parallel pow \parallel recordSig) \leq 2^{difficulty * count}$

10. Carol calculates PoW(*central*) and confirms that the output matches *nonce*.

If at any step an assertion fails, the Record is not valid and Carol does not accept it.

Record Broadcast

1. Bob derives the current Quorum by the Quorum Derivation protocol described in section 2.6.3.

2. Bob constructs a circuit, c_1, to a Mirror node m_1.

3. Bob asks for and receives from c_1 the $H(prevHash \parallel recordList \parallel consensusDocHash)$ hash and *pageSig* (the ed25519 signature on that hash) from each Quorum node.

4. Bob confirms via $V_{ed}(m, C)$ each *pageSig* and defines U as the largest set of Quorum nodes that have the same hash.

5. Bob randomly chooses a node n_1 from U and builds a circuit to it, c_2.
6. Bob uploads his Record through c_2 to n_1.
7. Bob waits up to Δs minutes for the next s flood iteration.
8. Bob uses c_1 to ask m_1 for any second-level domain name contained in his Record.
9. Bob has confirmation that his Record was accepted and processed by the Quorum if m_1 returns the Record he uploaded, assuming m_1 is not malicious and is synchronizing against a node other than n_1.
10. If m_1 does not return Bob's Record, he can either query another Mirror or repeat this procedure and broadcast to a different Quorum node. This choice is left up to the operator.

2.6.2 OnioNS Servers

Let Charlie be the name of an OnioNS Mirror.

Database Initialization

1. Charlie creates a empty Snapshot.
2. Charlie sets *originTime* to the number of seconds since the Unix epoch.
3. Charlie sets *recentRecords* to an empty array.
4. Charlie sets *fingerprint* to his Tor fingerprint.
5. Charlie sets $snapshotSig = S_{ed}(originTime \parallel recentRecords \parallel fingerprint, c)$ where c is Charlie's private Curve25519 NTor key.
6. Charlie begins listening for incoming Records.
7. Charlie creates an initial Page P_{curr}.
8. Charlie selects a previous Page P_{prev} to back-reference via the Page Selection protocol.

```
0 {
1       "prevHash": 0,
2       "recordList": [],
3       "consensusDocHash": "uU0nuZNNPgilLlLX2n2r+sSE7+N6U4DukIj3rOLvzek=",
4       "fingerprint": "2FC06226AE152FBAB7620BB107CDEF0E70876A7B",
5       "pageSig": "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/
            kBEpyXRSopzjoFs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh+
            QOnKl0fKBN7fqowjkQ3ktFkR0Vuox9WrrbNTMa4+
            up0Np52hlbKA3zSRz4fbR9NVlh6uuQ="
6 }
```

Figure 2.7: A sample empty Page.

9. Charlie sets $prevHash = H(P_{prev}(prevHash \parallel recordList \parallel consensusDocHash)$.

10. Charlie sets *recordList* to an empty array.

11. Charlie downloads from some remote source the consensus document *cd* issued on day $\lfloor \frac{q}{\Delta q} \rfloor$ at 00:00 GMT and authenticates it against the Tor authority public keys.

12. Charlie sets $consensusDocHash = H(cd)$.

13. Charlie sets *fingerprint* to his Tor fingerprint.

14. Charlie sets $pageSig = S_{ed}(H(prevHash \parallel recordList \parallel consensusDocHash), c)$ where c is Charlie's private Curve25519 NTor key.

Page Selection

The Page Selection protocol is used to select a Page in the midst of Page disagreements between current or past Quorum members. It relies on our original design assumption that the largest set of Quorum nodes with agreeing and valid Pages are acting honestly, so by extension the Page they maintain is honest as well and we can safely choose it. Let P_c be the Page that Charlie has selected.

1. Charlie obtains the set of Pages maintained by $Quorum_q$.

```
0 {
1       "originTime": 1426507551,
2       "recentRecords": [],
3       "fingerprint": "2FC06226AE152FBAB7620BB107CDEF0E70876A7B",
4       "snapshotSig": "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/
            kBEpyXRSopzjoFs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh+
            QOnKl0fKBN7fqowjkQ3ktFkR0Vuox9WrrbNTMa4+
            up0Np52hlbKA3zSRz4fbR9NVlh6uuQ="
5 }
```

Figure 2.8: A sample empty Snapshot.

2. Charlie obtains the consensus document *cd* issued on day $\lfloor \frac{q}{\Delta q} \rfloor$ at 00:00 GMT and authenticates it against the Tor authority public keys.

3. Charlie uses *cd* to calculate the Quorum via the Quorum Derivation protocol.

4. For each Page,

   (a) Charlie calculates $h = H(prevHash \parallel recordList \parallel consensusDocHash)$.

   (b) Charlie checks that *fingerprint* is a member of $Quorum_q$.

   (c) Charlie checks that $V_{ed}(h, C)$ returns true where C is *fingerprint*'s public Curve25519 key.

5. Charlie sorts the set of Pages by h, and constructs a 2D array of Pages that have the same h.

6. For each set of Pages with equal h,

   (a) Charlie checks that $consensusDocHash = H(cd)$.

   (b) Charlie checks each Record in *recordList* via the Record Validation protocol.

7. If the validation of a Page fails, Charlie removes it from the equal-h list.

8. Let P_c be chosen arbitrarily from the largest set of valid Pages with equal h.

In this way, Charlie need not preform a deep verification of all Pages from $Quorum_q$ in order to choose a Page.

Pagechain Validation

Assume that Charlie has obtained a complete Pagechain. Let P_c be an initial empty Page and let $f(P_1, P_2, q)$ accept two Pages and return true if $P_1 = P_2$ or if $q = 0$. For each q from the oldest available q to the most recent q,

1. Charlie chooses a Page P_{c2} from $Quorum_q$ via the Page Selection Protocol.
2. Charlie checks that $f(P_c, P_{c2}, q)$ returns true, or repeats step 1 to choose another Page from the next largest set of Pages that have equal h.

Synchronization

1. Charlie randomly selects a Quorum Candidate, R_j.
2. Charlie downloads R_j's Pagechain and the Pages used by each Quorum member for each Quorum, obtaining a 2D data structure at most L_P Pages long and L_Q Pages wide.
3. Charlie checks the downloaded Pagechain via the Pagechain Validation protocol. If it does not validate, Charlie picks another Quorum Candidate R_k, k/nej, and downloads the invalid Pages from R_k.
4. Charlie randomly selects a Quorum node Q_c.
5. Charlie periodically polls Q_c for the Pages of all other Quorum nodes.
6. When a set of Pages is available from Q_c, Charlie follows the Page Selection protocol to choose a Page that becomes the new Pagechain head.

Quorum Qualification

The Quorum is the OnioNS most trusted set of authoritative nodes. They have responsibility over the master Pagechain, are responsible for handling incoming Records from hidden service operators, and Mirrors poll them for their most recent Page. As such, the Quorum must be derived from the most reliable, capable, and trusted Tor nodes and more importantly Quorum nodes must be up-to-date Mirrors. These two requirements are crucial to ensuring the reliability and security of the Quorum.

The first criteria requires Tor nodes to demonstrate that they sufficient capabilities to handle the increase in communication and processing from with OnioNS protocols. Fortunately, Tor's infrastructure already provides a mechanism that can be utilized to demonstrate this requirement; Tor authority nodes assign flags to Tor routers to classify their capabilities, speed, or uptime history: these flags are used for circuit generation and hidden service infrastructure. Let Tor nodes meet the first qualification requirement if they have the Fast, Stable, Running, and Valid flags. As of February 2015, out of the $\approx$ 7,000 nodes participating in the Tor network, $\approx$ 5,400 of these node have these flags and complete the second requirement. [9]

To demonstrate the second criteria, the naïve solution is to simply ask nodes meeting the first criteria for their Page, and then compare the recency of its latest Page against the Pages from the other nodes. However, this solution does not scale well; Tor has $\approx$ 2.25 million daily users [9]: it is infeasible for any single node to handle queries from all of them. Instead, let each Mirror that meets the first criteria perform the following:

1. Charlie calculates $t = H(pc \parallel \lfloor \frac{m-15}{30} \rfloor)$ where pc is Charlie's Pagechain and m is the minute component of the time of day in GMT. Tor's consensus documents are published at the top of each hour; we manipulate m such that t is consistent at the top of each hour even with at most a 15-minute clock-skew.

2. Let Charlie convert t to base64 and truncate to 8 bytes.

3. Let Charlie include this new t in the Contact field in his relay descriptor sent to Tor authority nodes.

While ideally t could be placed inside a new field within the relay descriptor, to ease integration with existing Tor infrastructure and third-party tools we use the Contact field, a user-defined optional entry that Tor relay operators typically use to list methods of contact such as an email address. OnionNS would not be the first system to embed special information in the Contact field: PGP keys and BTC addresses commonly appear in the field, especially for high-performance routers.

Record Processing

A Quorum node Q_j listens for new Records from hidden service operators. When a Record is received, Q_j

1. Q_j rejects the Record if it does not validate according to the Record Validation protocol.

2. If it is a Create record, Q_j rejects it if any of its second-level domains already exist in Q_j's Pagechain.

3. If it is a Modify, Move, Renew or Delete Record, Q_j rejects it if either of the following are true:

   (a) It was not found in the Pagechain.

   (b) Its *pubHSKey* does not match the latest Record found in the Pagechain under its second-level domain names.

4. If Q_j has rejected the Record, Q_j informs Bob of this outcome and its reason.

5. If Q_j has not rejected the Record,

   (a) Q_j informs Bob that his Record was accepted.

   (b) Q_j merges the Record into its current Snapshot as shown in Figure 2.9.

   (c) Q_j regenerates *snapshotSig*.

```
0  {
1      "originTime": 1426507551,
2      "recentRecords": [{
3          "names": {
4              "example.tor": "exampleruyw6wgve.onion"
5          }
6          "contact": "AD97364FC20BEC80",
7          "timestamp": 1424045024,
8          "consensusHash": "uU0nuZNNPgilLlLX2n2r+sSE7+N6U4DukIj3rOLvzek=",
9          "nonce": "AAAABw==",
10         "pow": "4I4dzaBwi4AIZW8s2m0hQQ==",
11         "recordSig":     "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/
               kBEpyXRSopzjoFs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh+
               QOnKl0fKBN7fqowjkQ3ktFkR0Vuox9WrrbNTMa4+
               up0Np52hlbKA3zSRz4fbR9NVlh6uuQ=",
12         "pubHSKey": "MIGhMA0GCSqGSIb3DQEBAQUAA4GPADCBiwKBgQDE7CP/
               kgwtJhTTc4JpuPkvA7Ln9wgc+
               fgTKgkyUp1zusxgUAn1c1MGx4YhO42KPB7dyZOf3pcRk94XsYFY1ULkF2+
               tf9KdNe7GFzJyMFCQENnUcVXbcwLH4vAeiGK7R/nScbCbyc9LT+
               VE1fbKchTL1QzLVBLqJTxhR+9YPi8x+QIFAdZ8BJs="
13     }],
14     "fingerprint": "2FC06226AE152FBAB7620BB107CDEF0E70876A7B",
15     "snapshotSig": "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/
           kBEpyXRSopzjoFs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh+
           QOnKl0fKBN7fqowjkQ3ktFkR0Vuox9WrrbNTMa4+
           up0Np52hlbKA3zSRz4fbR9NVlh6uuQ="
16 }
```

Figure 2.9: A sample Create Record has been merged into an initially-blank Snapshot.

Flooding

Every Δs minutes, a burst of communication happens between Quorum nodes wherein Snapshots and Page signatures are flooded between them. This allows Quorum nodes to hear new Records and to know the status of the Pages maintained by their brethren. The exact timing of this protocol is dependent on the local system clock of each Quorum node. Let Q_j be a Quorum node with current Snapshot s_s, and at the Δs mark,

1. Q_j generates a new Snapshot, s_{s+1} via the Database Initialization protocol.

2. Q_j sets s_{s+1} to be the current Snapshot, used to collect new Records.

3. Q_j sets p_{old} to the current Page.

4. For every Q_k Quorum node, $k \neq j$,

   (a) Q_j asks Q_k for his snapshot.

   (b) Q_j merges the Records within Q_k's *recentRecords* into Q_j's current Page, as shown in Figure 2.10.

   (c) Q_j asks Q_k for its Page and sets the response as s_k.

   (d) Q_j calculates s_k's h value (see the Page Selection protocol) and if it matches a Page g already known to Q_j, Q_j archives an association between Q_k and g. In this case s_k's *pageSig* will validate g's h, allowing the width of the Pagechain to be grouped efficiently. Otherwise, Q_j archives s_k.

5. Q_j regenerates *pageSig*.

6. Q_j sends s_s when another Quorum node requests his Snapshot.

7. Q_j sends p_{old} when a Mirror or a Quorum node asks for his Page.

2.6.3 Tor Clients

Let Alice be a Tor client.

Quorum Derivation

1. Alice obtains a copy of the most recent consensus document, *cd*, published on day $\lfloor \frac{q}{\Delta q} \rfloor$ at 00:00 GMT.

2. Alice scans *cd* and constructs a list *qc* of Quorum Candidates of Tor routers that have the Fast, Stable, Running, and Valid flags and that are in the largest set of Tor routers that publish an identical time-based hash, as described in the Quorum Qualification protocol. She can construct *qc* in $\mathcal{O}(L_T)$ time.

3. Alice constructs $f = R(H(cd)$.

```
0  {
1      "prevHash": "4I4dzaBwi4AIZW8s2m0hQQ==",
2      "recordList": [{
3          "names": {
4              "example.tor": "exampleruyw6wgve.onion"
5          }
6          "contact": "AD97364FC20BEC80",
7          "timestamp": 1424045024,
8          "consensusHash": "uU0nuZNNPgilLlLX2n2r+sSE7+N6U4DukIj3rOLvzek=",
9          "nonce": "AAAABw==",
10         "pow": "4I4dzaBwi4AIZW8s2m0hQQ==",
11         "recordSig":     "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/
               kBEpyXRSopzjoFs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh+
               QOnKl0fKBN7fqowjkQ3ktFkR0Vuox9WrrbNTMa4+
               up0Np52hlbKA3zSRz4fbR9NVlh6uuQ=",
12         "pubHSKey": "MIGhMA0GCSqGSIb3DQEBAQUAA4GPADCBiwKBgQDE7CP/
               kgwtJhTTc4JpuPkvA7Ln9wgc+
               fgTKgkyUp1zusxgUAn1c1MGx4YhO42KPB7dyZOf3pcRk94XsYFY1ULkF2+
               tf9KdNe7GFzJyMFCQENnUcVXbcwLH4vAeiGK7R/nScbCbyc9LT+
               VE1fbKchTL1QzLVBLqJTxhR+9YPi8x+QIFAdZ8BJs="
13     }],
14     "consensusDocHash": "uU0nuZNNPgilLlLX2n2r+sSE7+N6U4DukIj3rOLvzek=",
15     "fingerprint": "2FC06226AE152FBAB7620BB107CDEF0E70876A7B",
16     "pageSig": "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/
           kBEpyXRSopzjoFs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh+
           QOnKl0fKBN7fqowjkQ3ktFkR0Vuox9WrrbNTMa4+
           up0Np52hlbKA3zSRz4fbR9NVlh6uuQ="
17 }
```

Figure 2.10: A Page containing an example Record. This Page is the result of the Snapshot in Figure 2.9 into an empty Page.

4. Alice uses f to randomly scramble qc.

5. The first $\min(\text{size}(qc), L_Q)$ routers are the Quorum.

Domain Query

Alice can optionally act as a resolver for other parties. Let Bob be a client and Faythe a third-party trusted by both Alice and Bob. Assume that Bob does not trust Alice, Bob has Faythe's public key, and that Bob has obtained a copy of the Tor consensus document

published on day $\lfloor \frac{i}{\Delta i} \rfloor$ at 00:00 GMT. Faythe has divided her bitset into Q sections, digitally signed each section, digitally signed the root hash of the Merkle tree, and send the signatures to Alice.

Bob enters "sub.example.tor" into his Tor Browser. As this TLD is not used by the Clearnet DNS, his client software directs the request to Alice. Bob must recursively resolve the .tor domain name into a .onion address and it is more efficient if Alice returns back to Bob all the necessary information that he needs to perform this resolution. Along with the domain name, Bob also sends to Alice one of the two verification levels, each providing progressively more verification that the record Bob receives is authentic, unique, and trustworthy. The default verification level is 0 for performance reasons.

At verification level 0, Alice first checks the hashtable bitset to confirm that the record exists. If it does, Alice then queries the AVL tree to find the latest record that contains the requested domain name. Alice resolves the requested domain and repeats this lookup up to eight times if the resulting destination uses the .tor TLD. Once a .onion TLD destination is encountered, Alice returns to Bob all records containing the intermediate and final destinations. If a lookup fails, Alice returns to Bob records containing any intermediary destinations, and the Faythe-signed section of the bitset. If the hash maps to a "1" bucket, Alice also returns the Faythe-signed root of the Merkle collision tree, the leaves of the branch containing neighbouring second-level domain names that alphabetically span the failed domain, and all other hashes at the top level of the branch in the Merkle tree.

Bob receives from Alice the data structures containing the domain name resolutions. Bob confirms the resolution path and confirms the validity, signature, and proof-of-work of each record in turn. Finally, Bob looks up the .onion hidden service in the traditional manner. If the resolution was unsuccessful, Bob also verifies the signed section of the bitset. If the domain maps to a "0", Bob's client software returns that the DNS lookup failed, otherwise Bob confirms that no such second-level domain name exists in the Merkle tree branch returned by Alice. Since the leaves of the branch alphabetically span the requested domain and would otherwise contain it if it existed, Bob knows that it does not

exist. Finally, Bob hashes the leaves to reconstructs the branch, and constructs the rest of the tree by combining the root of that branch with the other hashes returned by Alice at that level, and verifies the root hash against Faythe's signature. If this validates, Bob's software also informs the Tor Browser that DNS lookup failed. The resolution can also fail if the service has not published a recent hidden service descriptor to Tor's distributed hash table. End-to-end verification (relative to the authenticity of the hidden service itself) is complete when *pubHSKey* can be successfully used to encrypt the hidden service cookie and the service proves that it can decrypt *sec* as part of the hidden service protocol.

At verification level 1, in addition to returning all record and supplementary data structures in level 0, Alice also returns the page that contained each record and the *pageSig* field from Faythe's page. Since Alice and Faythe are maintaining a common page-chain, both Alice and Faythe will be signing the same data. Bob verifies the authenticity of the page against both Alice and Faythe's public keys and proceeds with his steps in level 0.

Onion Query

Bob may also issue a reverse-hostname lookup to Alice to find second-level domains that directly resolve to a given .onion hidden service address. This request is an Onion Query. Unlike on the Clearnet (under RFCs 1033 and 1912) not every .onion name has a corresponding domain name, so these queries may also fail. When Bob sends a Onion Query to Alice, Alice finds in her trie the record that contains a second-level .tor domain name that immediately maps to that .onion, and returns the domain name. Bob can optionally perform additional verification by issuing a Level 1 Domain Query on that domain name which should resolve to the original requested .onion hidden service address, a forward-confirmed reverse DNS lookup.

2.7 Optimizations

2.7.1 AVL Tree

A self-balancing binary AVL tree is used as a local cache of existing records. Its nodes

hold references to the location of records in a local copy of the page-chain and it is sorted by alphabetical comparison of second-level domain names. As a page-chain is a linear data structure that requires a $\mathcal{O}(n)$ scan to find a record, the $\mathcal{O}(n \log n)$ generation of an AVL tree cache allows lookups of second-level domain names to occur in $\mathcal{O}(\log n)$ time.

2.7.2 Hashtable Bitset

A hashtable bitset is a special and highly compact adaptation of a traditional hashtable. Unlike its AVL tree counterpart, its purpose to prove the non-existence of a record. As an extension of an ordinary hashtable, the hashtable bitset maps keys to buckets, but here we are interested in only tracking the existence of keys and have no need to store the keys themselves. Therefore each bucket in the structure is represented as a bit, creating a compact bitset of $\mathcal{O}(n)$ size. The hashtable bitset records a "1" if a second-level domain name exists, and a "0" if not. In the event of a hash collision, all second-level domains that map to that bucket should be added to an array list. Once the bitset is fully constructed, the array is sorted alphabetically and the resulted converted into the leaves of a Merkle tree.

Let Alice and a trusted authority Faythe maintain a common page-chain, and let Alice be a resolver for a third-party Bob. In the event that Bob asks Alice to resolve a domain name to a hidden service and Alice claims that the domain name does not exist, Bob must be able to verify the non-existence of that record. Demonstrating non-existence is a challenge often overlooked in DNS: even if existing records can be authenticated by Bob, Alice may still lie and claim a false negative on the existence of a domain name. Bob may choose to query Faythe to confirm non-existence, but this introduces additional load onto Faythe. Bob cannot easily determine the accuracy of Alice's claim without downloading all of the records and confirming for himself, but this is impractical in most environments.

The hashtable bitset efficiently resolves this issue. Faythe periodically generates a timestamped hashtable bitset for all existing domain names, digitally signs it, and publishes the result to Alice. As described in section **??**, Alice then includes this bitset along with any non-existence responses sent back to Bob. Since Bob knows and trusts Faythe, he can verify

Alice's claim by verifying the authenticity of the bitset and confirming that the domain he requested does not appear in the hashtable or in the Merkle tree in $\mathcal{O}(1)$ time on average. The hashtable also serves as an optimization to Alice: she can check the bitset for Bob's request in $\mathcal{O}(1)$ time, bypassing the $\mathcal{O}(logn)$ lookup if the second-level domain name does not exist.

Note that a Bloom filter with k hash functions could be used instead of a compact hashtable, but a Bloom filter would require sending up to k sections of buckets to the client. Therefore, we use a simple hashtable scheme, which is effectively a Bloom filter with $k = 1$.

2.7.3 Trie

A trie, also known as a digital tree, is used for efficiently resolving Onion Queries (section **??**). Each node in the trie corresponds to a base58 digit of the .onion hidden service address. These addresses are currently defined at 16 characters, so the trie has a maximum depth of 16 and up to 58 branches per node.

CHAPTER 3

ANALYSIS

We have designed OnionNS to meet our original design goals. Assuming that Tor circuits are a sufficient method of masking one's identity and location, hidden service operators can perform operations on their records anonymously. Likewise, a Tor circuit is also used when a client lookups a .tor domain name, just as Tor-protected DNS lookups are performed when browsing the Clearnet through the Tor Browser. OnionNS records are self-signed and include the hidden service's public key, so anyone — particularly the client — can confirm the authenticity (relative to the authenticity of the public key) and integrity of any record. This does not entirely prevent Sybil attacks, but this is a very hard problem to address in a distributed environment without the confirmation from a central authority. However, the proof-of-work component makes record spoofing a costly endeavour, but it is not impossible to a well-resourced attacker with sufficient access to high-end general-purpose hardware.

Without complete access to a local copy of the database a party cannot know whether a second-level domain is in fact unique, but by using an existing level of trust with a known network they can be reasonably sure that it meets the unique edge of Zooko's Triangle. Anyone holding a copy of the consensus document can generate the set of *quorum* node and verify their signatures. As the *quorum* is a set of nodes that work together and the *quorum* is chosen randomly from reliable nodes in the Tor network, OnionNS is a distributed system. Tor clients also have the ability to perform a full synchronization and confirming uniqueness for themselves, thus verifying that Zooko's Triangle is complete. Hidden service .onion addresses will continue to have an extremely high chance of being securely unique as long the key-space is sufficiently large to avoid hash collisions.

Just as traditional Clearnet DNS lookups occur behind-the-scenes, OnionNS Domain Queries require no user assistance. Client-side software should filter TLDs to determine

which DNS system to use. We introduce no changes to Tor's hidden service protocol and also note that the existence of a DNS system introduces forward-compatibility: developers can replace hash functions and PKI in the hidden service protocol without disrupting its users, so long as records are transferred and OnionNS is updated to support the new public keys. We therefore believe that we have met all of our original design requirements.

3.1 Security

3.1.1 Quorum-level Attacks

The quorum nodes hold the greatest amount of responsibility and control over OnionNS out of all participating nodes in the Tor network, therefore ensuring their security and limiting their attack capabilities is of primary importance.

Passively Malicious Quorum

In section **??**, we assumed that an attacker, Eve, already controls a fraction of Tor routers. For a dynamic network size, a fixed fraction of dishonest nodes, and a fixed quorum size, every time a quorum is selected there is a probability that more than half of the quorum is dishonest and is colluding together. If this occurs, records may be dropped rather than archived. *Here we explore the optimal quorum rotation rate, given the balance between high overhead and high security risk if the quorum is rotated quickly, and long-term implications and possible stability issues associated with very slow rotation.*

Active Malicious Quorum

If Eve controls some Tor nodes (who may be assumed to be colluding with one another), the attacker may desire to include their nodes in the quorum for malicious manipulation, passive observation, or for other purposes. Alternatively, Eve may wish to exclude certain legitimate nodes from inclusion in the quorum. In order to carry out either of these attacks, Eve must have the list of qualified Tor nodes scrambled in such a way that the output is pleasing to Eve. Specifically, the scrambled list must contain at least some of Eve's malicious

nodes for the first attack, or exclude the legitimate target nodes for the second attack. We initialize Mersenne Twister with a 384-bit seed, thus Eve can find k seeds that generates a desirable scrambled list in 2^{192} operations on average, or 2^{384} operations in the worst case. The chance of any of those seeds being selected, and thus Eve successfully carrying out the attack, is thus $\frac{2^{384}}{k}$.

Eve may attempt to manipulate the consensus document in such a way that the SHA-384 hash is one of these k seeds. Eve may instruct her Tor nodes to upload a custom status report to the authority nodes in an attempt to maliciously manipulate the contents of the consensus document, but SHA-384's strong preimage resistance and the unknown state and number of Tor nodes outside Eve's control makes this attack infeasible. The time to break preimage resistance of full SHA-384 is still 2^{384} operations. This also implies that Eve cannot determine in advance the next consensus document, so the new quorum cannot be predicted. If Eve has compromised at least some of the Tor authority nodes she has significantly more power in manipulating the consensus document for her own purposes, but this attack vector can also break the Tor network as a whole and is thus outside the scope of our analysis. Therefore, the computation required to maliciously generate the quorum puts this attack vector outside the reach of computationally-bound adversaries.

OnionNS and the Tor network as a whole are both susceptible to Sybil attacks, though these attacks are made significantly more challenging by the slow building of trust in the Tor network. Eve may attempt to introduce large numbers of nodes under her control in an attempt to increase her chances of at least one of the becoming members of the *quorum*. Sybil attacks are not unknown to Tor; in December 2014 the black hat hacking group LizardSquad launched 3000 nodes in the Google Cloud in an attempt to intercept the majority of Tor traffic. However, as Tor authority nodes grant consensus weight to new Tor nodes very slowly, despite controlling a third of all Tor nodes, these 3,000 nodes moved 0.2743 percent of Tor traffic before they were banned from the Tor network. The Stable and Fast flags are also granted after weeks of uptime and a history of reliability. As nodes must have these flags to be qualified as a *quorum candidate*, these large-scale Sybil attacks

are financially demanding and time-consuming for Eve.

3.1.2 Non-existence Forgery

As we have stated earlier, falsely claiming a negative on the existence of a record is a problem overlooked in other domain name systems. One of the primary challenges with this approach is that the space of possible names so vast that attempting to enumerate and digitally sign all names that are not taken is highly impractical. Without a solution, this weakness can degenerate into a denial-of-service attack if the DNS resolver is malicious towards the client. Our counter-measure is the highly compact hashtable bitset with a Merkle tree for collisions. We set the size of the hashtable such that the number of collisions is statistically very small, allowing an efficient lookup in $\mathcal{O}(1)$ time on average with minimal data transferred to the client.

3.1.3 Name Squatting and Record Flooding

An attacker, Eve, may attempt a denial-of-service attack by obtaining a set of names for the sole purpose of denying them to others. Eve may also wish to create many name requests and flood the *quorum* with a large quantity of records. Both of these attacks are made computationally difficult and time-consuming for Eve because of the proof-of-work. If Eve has access to large computational resources or to custom hardware she may be able to process the PoW more efficiently than legitimate users, and this can be a concern.

The proof-of-work scheme is carefully designed to limit Eve to the same capabilities as legitimate users, thus significantly deterring this attack. The use of scrypt makes custom hardware and massively-parallel computation expensive, and the digital signature in every record forces the hidden service operator to resign the fields for every iteration in the proof-of-work. While the scheme would not entirely prevent the operator from outsourcing the computation to a cloud service or to a secondary offline resource, the other machine would need the hidden service private key to regenerate *recordSig*, which the operator can't reveal without compromising his security. However, the secondary resource could perform the scrypt computations in batch without generating *recordSig*, but it would always perform

more than the necessary amount of computation because it would could not generate the SHA-384 hash and thus know when to stop. Furthermore, offloading the computation would still incur a cost to the hidden service operator, who would have to pay another party for the consumed computational resources. Thus the scheme always requires some cost when claiming a domain name.

3.2 Performance

bandwidth, CPU, RAM, latency for clients to be determined...

3.2.1 Load

demand on participating nodes to be determined...

Unlike Namecoin, OnionNS' *page*-chain is of L days in maximal length. This serves two purposes:

1. Causes domain names to expire, which reduced the threat of name squatting.
2. Prevents the data structure from growing to an unmanageable size.

3.3 Reliability

Tor nodes have no reliability guarantee and may disappear from the network momentarily or permanently at any time. Old *quorums* may disappear from the network without consequence of data loss, as their data is cloned by current *mirrors*. So long as the *quorum* nodes remain up for the Δi days that they are active, the system will suffer no loss of functionality. Nodes that become temporarily unavailable will have out-of-sync *pages* and will have to fetch recent records from other *quorum* nodes in the time of their absence.

REFERENCES

[1] D. Chaum, "Untraceable electronic mail, return addresses and digital pseudonyms," in *Secure electronic voting*. Springer, 2003, pp. 211–219.

[2] M. Edman and B. Yener, "On anonymity in an electronic society: A survey of anonymous communication systems," *ACM Computing Surveys (CSUR)*, vol. 42, no. 1, p. 5, 2009.

[3] P. Syverson, "A peel of onion," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 123–137.

[4] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," DTIC Document, Tech. Rep., 2004.

[5] P. F. Syverson, D. M. Goldschlag, and M. G. Reed, "Anonymous connections and onion routing," in *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. IEEE, 1997, pp. 44–54.

[6] M. G. Reed, P. F. Syverson, and D. M. Goldschlag, "Anonymous connections and onion routing," *Selected Areas in Communications, IEEE Journal on*, vol. 16, no. 4, pp. 482–494, 1998.

[7] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker, "Low-resource routing attacks against tor," in *Proceedings of the 2007 ACM workshop on Privacy in electronic society*. ACM, 2007, pp. 11–20.

[8] L. Overlier and P. Syverson, "Locating hidden servers," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 15–pp.

[9] T. T. Project, "Tor metrics," https://metrics.torproject.org/, 2015, accessed 4-Feb-2015.

[10] S. Landau, "Highlights from making sense of snowden, part ii: What's significant in the nsa revelations," *Security & Privacy, IEEE*, vol. 12, no. 1, pp. 62–64, 2014.

[11] R. Plak, "Anonymous internet: Anonymizing peer-to-peer traffic using applied cryptography," Ph.D. dissertation, TU Delft, Delft University of Technology, 2014.

[12] D. Lawrence, "The inside story of tor, the best internet anonymity tool the government ever built," *Bloomberg BusinessWeek*, January 2014.

[13] D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker, "Shining light in dark places: Understanding the tor network," in *Privacy Enhancing Technologies*. Springer, 2008, pp. 63–76.

[14] Z. Ling, J. Luo, W. Yu, X. Fu, W. Jia, and W. Zhao, "Protocol-level attacks against tor," *Computer Networks*, vol. 57, no. 4, pp. 869–886, 2013.

[15] I. Goldberg, "On the security of the tor authentication protocol," in *Privacy Enhancing Technologies*. Springer, 2006, pp. 316–331.

[16] I. Goldberg, D. Stebila, and B. Ustaoglu, "Anonymity and one-way authentication in key exchange protocols," *Designs, Codes and Cryptography*, vol. 67, no. 2, pp. 245–269, 2013.

[17] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *Public Key Cryptography-PKC 2006*. Springer, 2006, pp. 207–228.

[18] L. Xin and W. Neng, "Design improvement for tor against low-cost traffic attack and low-resource routing attack," in *Communications and Mobile Computing, 2009. CMC'09. WRI International Conference on*, vol. 3. IEEE, 2009, pp. 549–554.

[19] S. Nicolussi, "Human-readable names for tor hidden services," 2011.

[20] J. Li, T. Isobe, and K. Shibutani, "Converting meet-in-the-middle preimage attack into pseudo collision attack: Application to sha-2," in *Fast Software Encryption*. Springer, 2012, pp. 264–286.

[21] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," in *Cryptographic Hardware and Embedded Systems–CHES 2011*. Springer, 2011, pp. 124–142.

[22] C. Percival, "Stronger key derivation via sequential memory-hard functions," *Self-published*, pp. 1–16, 2009.

[23] C. Percival and S. Josefsson, "The scrypt password-based key derivation function," 2012.

[24] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.