THE ONION NAME SYSTEM:
TOR-POWERED DISTRIBUTED DNS
FOR TOR HIDDEN SERVICES

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

PUBLIC ABSTRACT

Jesse M. Victors

The Tor network is a third-generation onion router that aims to provide private and anonymous Internet access to its users. In recent years its userbase, network, and community has grown significantly in response to revelations of national and global electronic surveillance, and it remains one of the most popular anonymity networks in use today. Tor also provides access to anonymous servers known as hidden services – servers of unknown location and ownership that may provide websites, chat services, or an electronic dead drop. These hidden services can be accessed through any Tor-powered web browser but they suffer from usability challenges due to the algorithmic generation of their addresses.

In response to this difficulty, in this work we introduce the Onion Name System (OnioNS), a privacy-enhanced distributed DNS that allows hidden service operators to select a globally-unique domain name for their service. We construct OnioNS as an optional backwards-compatible plugin for Tor on top of existing hidden service infrastructure and utilize the existing Tor network, which minimizes our assumptions and simplifies our threat model. Additionally, OnioNS allows clients to verify the authenticity or non-existence of domain names with minimal networking costs without introducing any central authority.

The authors would like to thank Mark Ellzey, Yawning Angel, and Nick Mathewson for their assistance with libevent and Tor technical support; Sarbajit Mukherjee for his commentary; and the Tor community for their continued support.

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. [3] 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. [4] [5]

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. [1] 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 [6] [7] 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. [5] 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

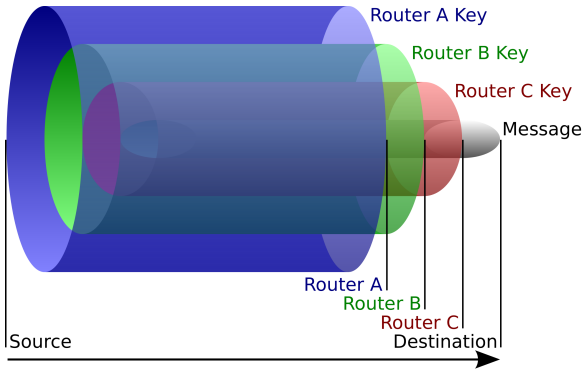

Figure 1.1: An example cell and message encryption in an onion routing scheme. Each router "peals" off its respective layer of encryption; the final router exposes the final destination.

static network topology, and most notably, introduced the ability to mate two circuits at a common router 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. [5]

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 [1] 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: [5] [1]

- **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 router 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 session 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. [5] Tor abandoned this in favor of isolating circuits from each other inside the network, although it recycles TCP/IP links between routers.

- **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 router (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 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. [8] [9]

- **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 sanitization, security, and privacy enhancements not included in native Firefox. The browser also utilizes the Electronic Frontier Foundation's HTTPS Everywhere extension to re-write HTTP web requests into HTTPS whenever possible, providing an additional encryption layer that hides web traffic from exit routers.

- **Directory servers:** Tor introduced a set of trusted directory servers, called directory authorities, to collect, digitally sign, and distribute network information such as the IP addresses and public keys of onion routers. Onion routers mirror the network information from the directories, distributing the bandwidth 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 authority is independently maintained in multiple locations and jurisdictions, reducing the likelihood of an attacker compromising all of them. [5] We describe the contents and format of this network information 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 introduced 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. [1] 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 directory authorities and 6,600 onion routers in

83 countries. [10] 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". [11] [12] In 2014, BusinessWeek claimed that Tor was "perhaps the most effective means of defeating the online surveillance efforts of intelligence agencies around the world." [13]

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. [1]

1.2.2 Circuit Construction

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.

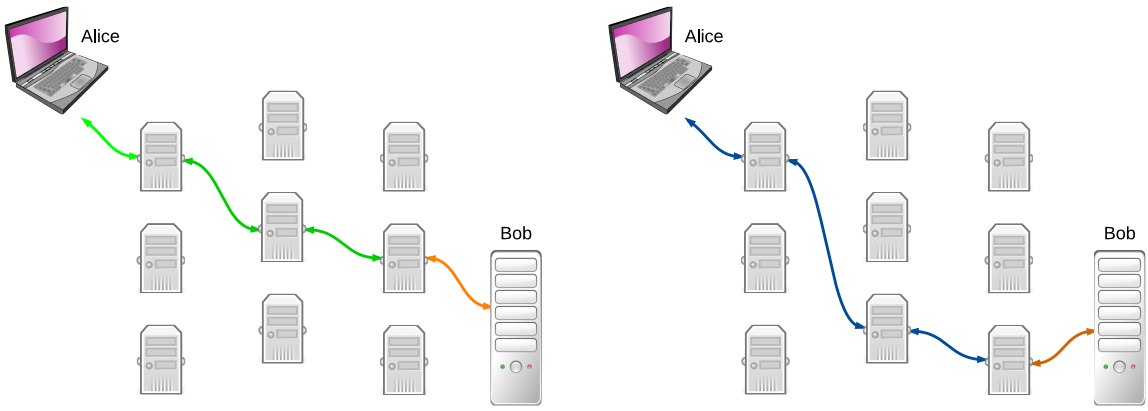

Figure 1.2: Alice communicates privately to Bob through a Tor circuit. Her communication path consists of three routers, an entry, middle, and exit. Although Bob's identity and location is known to Alice, the Tor circuit prevents Bob from knowing Alice's identity or location. At a later time, Alice may construct a different circuit to Bob, giving her a new identity from Bob's perspective. Each encrypted Tor link is shown in green, the final connection from the exit to Bob, shown in orange, is optionally encrypted.

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 defence against traffic analysis, her message is padded into equally-sized Tor *cells* as it travels through the network. [14]

TAP

The Tor Authentication Protocol (TAP) is a central exchange in Tor's networking infrastructure. If a Tor router R_i is known to Alice, TAP allows Alice to authenticate R_i and to enables them to negotiate an ephemeral session encryption key while preserving Alice's anonymity. TAP thus provides perfect forward secrecy in Tor circuits and defends against spoofing attacks.

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 directory authorities, 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.

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]

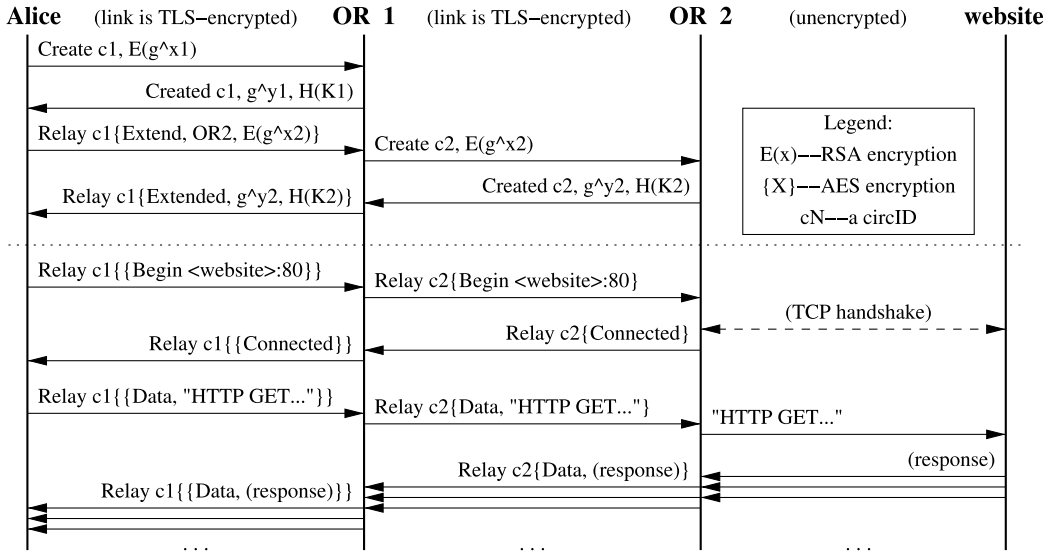

Figure 1.3: The Tor Authentication Protocol negotiated over two onion routers. Following circuit construction, Alice sends a HTTP request to the target server, which is encrypted by each router as it travels back up the circuit. [1] [2]

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 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

As previously mentioned, early mixnets and onion routers either assumed a static network topology or flooded updates across the network. By contrast, Tor's network is maintained by a small set of semi-trusted directory authorities. Periodically, Tor routers

upload digitally signed "descriptors" to these authorities. A descriptor may contain essential routing numbers, router capabilities, cryptographic keys, bandwidth history, or other information.

Each directory authority maintains an long-term authority key (distinct from its normal identity key if it is a Tor router) and a medium-term signing key. Periodically, each directory authority

1. Aggregates the descriptors into a single "status vote" document.
2. Signs its vote with its signing key.
3. Exchanges its vote and signature with all other authorities.
4. Computes a single network status consensus from all the other voting documents.
5. Signs the network consensus and exchanges the signature with all other authorities.

Once this is complete, the consensus is published and is available for download. If clients have knowledge of the Tor network, they may download the more recent consensus from the directory-mirroring routers. By this system, new routers or changes to existing routers can be propagated to all parties within a very short timeframe. Although routers can optionally publish additional non-essential descriptors, clients and routers typically only need the essential descriptors containing routing information, directory signing keys, and router keys. We discuss the documents containing these descriptors below: [18] [19]

cached-certs

The *cached-certs* document contains the long-term authority identity keys and the medium-term signing keys from each directory authority. The Tor source includes the long-term keys, so all parties can verify the authenticity of the signing keys and in turn the descriptors signed by them. They will believe router descriptors if more than half of the authorities have signed it. Each certificate contains the following fields:

- **fingerprint:** The SHA-1 hash of the identity key.

- **dir-key-published:** The time in UTC when the keys were last published.
- **dir-key-expires:** The time in UTC when the signing keys expire.
- **dir-identity-key:** The identity key, typically a 3072-bit RSA key.
- **dir-signing-key:** The signing key, typically a 2048-bit RSA key.
- **dir-key-crosscert:** The signature of the identity key, made using the signing key.
- **dir-key-certification:** The signature of the above fields, made using the identity key.

cached-microdesc-consensus

The *cached-microdesc-consensus* document contains network status information. The document includes a header and then a list of condensed descriptors from each router, called microdescriptors. The header includes the following fields:

- **valid-after** (VA), **fresh-until** (FU), and **valid-until** (VU). Three timestamps in UTC, VA $<$ FU $<$ VU. VA and VU specifies the earliest and latest time that these descriptors are valid, respectively. All three values are chosen such that two consensuses overlap: consensus_x will be considered fresh until consensus_{x+1} becomes valid, and then consensus_x expires when consensus_{x+1} is no longer fresh.
- **client-versions** and **server-versions:** An ascending list of recommended Tor versions for clients and routers, respectively.
- A list of directory authorities, each containing:
  - **dir-source:** The authority's nickname, fingerprint, IP address, onion routing port, and directory port.
  - **contact:** Optional contact information for the authority operator.
  - **vote-digest:** The hash of the authority's status vote document.

Following the header, each microdescriptor contains:

- **r:** The router's nickname, fingerprint, time of last restart, IP address, onion routing port, and directory port.
- **m:** The SHA-256 hash of the router's microdescriptor. This also includes its entries in the *cached-microdescs* document (discussed below).
- **s:** A list of the router's status flags, as given by the directory authorities. Common examples include Running, Valid, Fast, Guard, Stable, and Exit.
- **v:** The version of the Tor software that the router is running, as reported by the router.
- **w:** The estimated bandwidth that this router is capable of. This value is determined by speed tests from bandwidth authorities, who are a subset of the directory authorities.

cached-microdescs

The *cached-microdescs* document contains cryptographic keys from each Tor router. Each entry contains:

- **onion-key:** The router's public RSA key, primarily used for TAP authentication.
- **ntor-onion-key:** The router's public Curve25519 key, primarily used for NTor.
- **family:** The fingerprint of routers also under the operator's administration; Tor clients will not construct circuits through any routers that have the same family or that are in the same /16 IPv4 block.
- **id:** The router's fingerprint.

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. [20]

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. [8] [9]

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. 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. [10]

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 3.1).

- We described a new distributed, publicly confirmable, and partially self-healing transactional database.
- We provide OnioNS as a Tor plugin and 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

PROBLEM STATEMENT

2.1 Assumptions and Threat Model

OnioNS' basic design and protocols rely on several assumptions and expected threat vectors.

1. Let Alice is a Tor user and let Bob be a recipient. If Alice constructs a three-router Tor circuit via NTor and sends a message m to Bob, we assume that the Tor circuit preserves Alice's privacy and provides her with anonymity from Bob's perspective. Namely, we assume that out of Alice's identity, location, and knowledge of Bob, an outside attacker Eve can obtain at most one, and that Bob can know no more about Alice than the contents of m. The exception to this is if Alice chooses to disclose her identity only to Bob, but then Bob does not know Alice's location. We assume that neither Bob nor Eve can force Alice into breaking her privacy involuntarily. The protection of a Tor circuit relies on Tor's assumption that Eve only has access to some Tor traffic; no defence is made by Tor nor by OnioNS to defend against a global adversary.

2. Adversaries cannot break standard cryptographic primitives.

   (a) We assume that they cannot efficiently break AES ciphers, SHA-384 hashes, RSA-1024, Curve25519 ECDHE, Ed25519 digital signatures, or have hardware-level attacks against the scrypt key derivation function.

   (b) We assume that they maintain no backdoors or other software breaks in the Botan or the OpenSSL implementations of the primitives.

(c) We assume that they are not capable of breaking cryptographic protocols built on the primitives such as TAP and NTor.

3. We assume that not all Tor nodes are honest. We assume that at least some Tor nodes are run by malicious operators, curious researchers, experimenting developers, or government organizations. They may also be wiretapped, exploited, broken into, or become unavailable. This assumption is also made by Tor's developers and therefore we must conclude that any new functionality added to existing Tor nodes must also fall under their assumption. However, we assume that the majority of Tor nodes are honest and reliable. We consider this reasonable because it is a prerequisite for the security of Tor circuits.

4. We assume that the percentage of dishonest routers within the Tor network does not increase in response to the inclusion of OnionNS into Tor infrastructure. This assumption simplifies our threat model analysis but we consider it realistic because Tor traffic is purposely kept secret and is therefore much more valuable to Eve. OnioNS uses public data structures so we don't consider the inclusion of OnioNS a motivating factor to Eve. Moreover, we assume that Eve cannot predict in advance the identities of the Tor routers that act as authorities for OnioNS.

5. Let C be the set of Tor nodes that have the Fast and Stable flags and let Q be an M-sized set chosen randomly from C. Q may be under the influence of one or more adversaries who may cause subsets of Q to collude, but our final assumption is that the largest subset of Q is acting honestly according to specifications.

2.2 Design Objectives

Tor's high security environment introduces a distinct set of challenges that must be met by additional functionality. Privacy and anonymity are of paramount importance. Here we enumerate a list of requirements for any security-enhanced DNS applicable to Tor hidden services. In Chapter we analyse several existing prominent naming systems and show how

these systems do not meet these requirements. In Chapters and we demonstrate how we overcome them with OnioNS.

1. **The system must support anonymous registrations.** It must be hard for any party to infer the identity or location of the registrant, unless the registrant chooses to disclose that information. Hidden services are privacy-enhanced servers and the registration of a domain name should not compromise that privacy.

2. **The system must support privacy-enhanced lookups.** The resolver should not be able to identify a client nor track their lookups. In other words, clients should ideally have anonymity and be indistinguishable from other clients from the perspective of a resolver.

3. **Clients must be able to authenticate registrations.** The users of the system must be able to verify that the information they received from the service has not been forged and is authenticate relative to the authenticity of the server they will connect to.

4. **Domain names must be provably or have a near-certain chance of being unique.** Any domain name of global scope must point to at most one server. In the case of naming systems that generate names via cryptographic hashes, the domain name key-space must be of sufficient length to be remain resistant to at least collision and second pre-image attacks.

5. **The system must be distributed.** Systems with root authorities have distinct disadvantages compared to distributed networks: specifically, central authorities have absolute control over the system and root security breaches could easily compromise the integrity of the entire system. Root authorities may also be able to easily compromise user privacy or may not allow anonymous registrations, although this is system dependent. For these reasons, naming systems with central authorities can safely be considered dangerous and ill-suited for hidden services.

6. **The system must be simple and relatively easy to use.** It should be assumed that users are not security experts or have technical backgrounds. Tor's developers go to great lengths to ensure that Tor tools are straightforward, and the success of Tor's low-latency onion routing model demonstrates the effectiveness of convenience within high-security settings. Any naming system used by Tor must be equivalently simple, resolve protocols with minimal input from the user, and hide non-essential details.

7. **The system must be backwards compatible with existing protocols.** Just as the Internet's DNS did not introduce any changes in the TCP/IP layer, naming systems for Tor must preserve the original Tor hidden service protocol. DNS is optional, not required.

Additionally, we specify several performance-enhancing objectives, although these are not requirements.

1. **The system should not require clients to download the entire database.** It can safely assumed that in most realistic environments clients have neither the network capacity nor the storage to hold the system's database. While they may choose to download the database, it should not be required for normal functionality or to meet any of the above objectives.

2. **The system should not introduce significant burdens to the clients.** Despite the rapid and exponential growth of consumer-grade hardware, in most environments not every client is on a high-end machine. Therefore the system should not burden user computers with significant computation or memory demands.

3. **The system should have low latency** and resolve lookup queries within a reasonable amount of time.

CHAPTER 3

CHALLENGES

3.1 Zooko's Triangle

Our primary objective with OnioNS is to provide human-meaningful domain names Tor hidden services, but we list distributed and securely unique domain names in our design requirements. Achieving all three objectives is not easy; a naming scheme can use a central root zone or authority to ensure that meaningful domains remain unique but then it is not distributed, it can achieve a distributed nature by generating domain names with cryptographic hash function but then domains are no longer human-meaningful, or it can allow peers to provide meaningful names to each other but then these names are not guaranteed to be globally unique. This problem is illustrated in Figure 3.1 and summarized by Zooko's Triangle, a conjecture proposed by Zooko Wilcox-O'Hearn in late 2001. The conjecture states a persistent naming system can achieve at most two of these properties: it can provide unique and meaningful names but not be distributed, it can be distributed and provide unique names that are not meaningful, or it can be distributed and provide meaningful names that are not guaranteed to be unique. [21] [22]

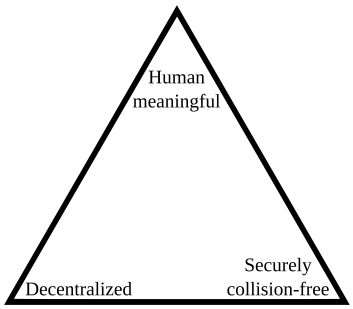

Figure 3.1: Zooko's Triangle.

Petnames are systems that achieve all three properties of Zooko's Triangle. Some examples of naming systems that achieve only two of these properties include:

- **Securely unique and human-meaningful** — Clearnet domain names are memorable and provably collision free, but they use DNS with a hierarchical structure and central authorities under the jurisdiction of ICANN.
- **Decentralized and human-meaningful** — Human names and nicknames are legal and social labels for each other, but we provide no protection against name collisions.
- **Securely unique and decentralized** — Tor hidden service .onion domains, PGP keys, and Bitcoin/Namecoin addresses use the large key-space and the collision-free properties of cryptographic hash algorithms to ensure uniqueness, but do not use meaningful names.

We consider the securely unique and human-meaningful properties the most important objectives, and like Namecoin (discussed in section 4.3) we provide a mechanism that enables the network to agree on a common database and largely prevents it from fragmenting. Additionally, all participants can confirm for themselves the integrity of the database and the uniqueness of domain names contained within it. This holds the securely unique property within the distributed environment.

3.2 Non-existence Verification

In our design requirements we specify that clients of a naming system should be able to verify the authenticity of domain names. On the Internet, this is achieved through SSL certificates: clients first receive from the destination server a digital certificate, they verify that the certificate matches the domain name they requested to their DNS resolver, and finally they check the certificate against Certificate Authorities via a Chain of Trust. Of equal importance, however, is the capability to verify the non-existence of domain names. Specifically a resolver may falsely claim that a domain name cannot be resolved because it does not exist, but a client has no mechanism by which to verify this claim besides changing

resolvers. This is a weakness often overlooked in other DNSs and resolving this problem is not easy. However, we in section 5.7.3 we introduce a data structure and a protocol that allows a resolver to proof non-existence in constant time.

CHAPTER 4

EXISTING WORKS

Todo: this chapter will be rewritten and expanded to include other DNSs

Several workarounds exist that attempt to alleviate the issue, with one used in practice, but none are fully successful. Other DNS systems already exist, including some that are distributed, but they are either not applicable, or their design makes it extremely challenging to integration into the Tor environment. Two primary problems occur when attempting to use existing DNS systems: being able to prove the correlation of ownership between the DNS name and the hidden service, and the leakage of information that may compromise the anonymity of the hidden service or its operator. As hidden services are only known by their public key and introduction points, it is not easy to use only the hidden service descriptor to prove ownership, and there are privacy and security issues with requiring any more information. Due to these problems, no existing works have been yet integrated into the Tor environment, and the one workaround that is used in practice remains only partially successful.

4.1 Encoding Schemes

In an attempt to address the readability and the memorability of hidden service domain names, two changes to the encoding of domain names have been proposed, with one used in practice.

Shallot, originally created by an anonymous developer sometime between 2004 and 2010, is an application which uses OpenSSL to generate many RSA hidden service keys in an attempt to find one that has a desirable hash, such as one that begins with a meaningful noun. [23] By brute-forcing the domain key-space, Shallot will eventually find a domain that is meaningful and partially memorable to the hidden service operator. Shallot was used by

Blockchain.info (blockchainbdgpzk.onion), by Facebook (facebookcorewwwi.onion), and by many other hidden services in an attempt to make their domain name appear less random. However, Shallot only partially successful because the size of the domain key-space is too large to be fully brute-forced in any reasonable length of time. For example, on a 1.5 GHz processor, Shallot is expected to take approximately 25 days to find a key whose hash contains 8 custom letters out of the 16 total. [23] Tor Proposal 224 makes this solution even worse as it suggests 32-byte domain names which embed an entire ECDSA hidden service key in base32. [24] Although prefixing the domain name with a meaningful word helps identify a hidden service at a glance, it does nothing to alleviate the logistic problems of entering a hidden service domain name, including the remaining random characters, manually into the Tor Browser.

A different encoding scheme was proposed in 2011 by Simon Nicolussi, who suggested that encoding the key hash as a series of words, using a dictionary known in advance by all parties. The list of words would then represent the domain name, rather than base58. While this scheme would improve the memorability of hidden services, the words cannot all be chosen by the hidden service operator, and brute-forcing with Shallot would again only be partially successful due to the large key-space. Therefore this solution could be used to generate some words that relate to the hidden service in a meaningful way, but this scheme is only a partial solution. [20]

These schemes do not change the underlying hidden service protocol, they just attempt to increase the readability of the domain names in Tor Browser. Compared against our original requirements, they meet the anonymity for registration and lookups due to Tor circuits, the confirmability because the domain is the hash of the public key, the uniqueness of domain names due to the collision-free property of SHA-1, the distributed requirement by way of the distributed hashtable stored throughout the Tor network, and resistance to malicious modifications because of the strong association between domain names and the hidden service key. However, it fails to meet the simplicity requirement because although hidden service domain names are entered in the traditional manner into the Tor Browser, the

domain names are not entirely human-meaningful nor memorable. The domain names continue to suffer from usability problems, only partially alleviated by these encoding schemes. These workarounds do not introduce any new DNS systems or change the existing Tor hidden service protocol in any way that allows for customized domain names, but these partial solutions are worthy of mention nevertheless.

4.2 Internet DNS

The Internet Domain Name Service (DNS) was originally designed in 1983 as a hierarchical naming system to link domain names to Internet Protocol (IP) addresses and translate one to the other. IP addresses specify the location of a computer or device on a network and domain names identify that resource. Domain names therefore operate as an abstraction layer, allowing servers to be moved to a different IP address without loss of functionality. The names consist of a top-level domain (TLD) that is prefixed by a sequence of labels, delimited by dotes. The labels divide the DNS system into zones of responsibility, where each label can be unique within that zone, but not necessarily across different zones. Each label can consist of up to 63 characters and the domain names can be up to 253 characters in total length. In contrast to IP addresses, domain names are human-meaningful and easily memorized, so DNS is a crucial component to the usability of the Internet.

The Internet DNS system suffers from several significant shortcomings that make it inappropriate for use by Tor hidden services. First, responses to DNS queries are not authenticated by digital signatures, so spoofing by a MITM attack is possible. Secondly, queries and responses are not encrypted, so it is easy for anyone wiretapping port 53 to correlate end-users to their activities, even if the communication to those websites is protected by TLS/HTTPS. Third, it suffers from DNS cache poisoning, in which an attacker pretends to be an authoritative server and sends incorrect or malicious records to local DNS resolvers. Finally, owning a TLD specifically for Tor hidden services (such as .tor) is prohibitively expensive. While DNS looks traditionally go through the Tor circuit and are resolved by Tor exit nodes, the shortcomings of the Internet DNS system make it unsuitable for our purposes.

The traditional Internet DNS system meets only a few of our requirements; registrations require a significant of identifiable and geographic information and thus are far from anonymous, lookups occur without encryption or signatures and are therefore neither anonymous nor privacy-enhanced, registrations are by default not publically confirmable but can be made so through use of an expensive SSL certificate from a central authority, the system is zone-based but is managed by centralized organizations and is therefore only partially distributed, and while it is very simple to use, extremely popular, and backwards compatible with TCP/IP, a Tor-specific TLD does not exist and falsifications of registrations is possible in a number of different ways. For its many security issues we therefore dismiss it as a possible solution.

4.3 Namecoin

Namecoin (NMC) is a decentralized peer-to-peer information registration and transfer system, developed by Vincent Durham in early 2011. It was the first fork of Bitcoin and as such inherits most of Bitcoin's design and capabilities. Like Bitcoin, a digital cryptocurrency created by pseudonymous developer Satoshi Nakamoto in 2008, Namecoin holds name records and transactions in a public ledger known as a blockchain. Users may hold Namecoins, and may prove ownership and authorize transactions with their private secp256k1 ECDSA key. Each transaction is a transfer of ownership of a certain amount of NMC from one public key to another, and as all transactions are recorded into the blockchain by the Namecoin network, all Namecoins can be traced backwards to the point of origin. Purchasing a name-value pair, such as a domain name, consumes 0.01 Namecoins, thus adding value and some expense to names in the system. Durham added two rules to Namecoin not present in Bitcoin: names in the blockchain expire after 36,000 blocks (about every 250 days) unless renewed by the owner and no two unexpired names can be identical. Namecoin domain names use the .bit TLD, which is not in use by ICANN.

The blockchain data structure is Nakamoto's novel answer to the problem of ensuring agreement of critical data across all involved parties. This prevents the double-spending of Bitcoins, or in Namecoin's case the prevention of duplicate names. Starting from an

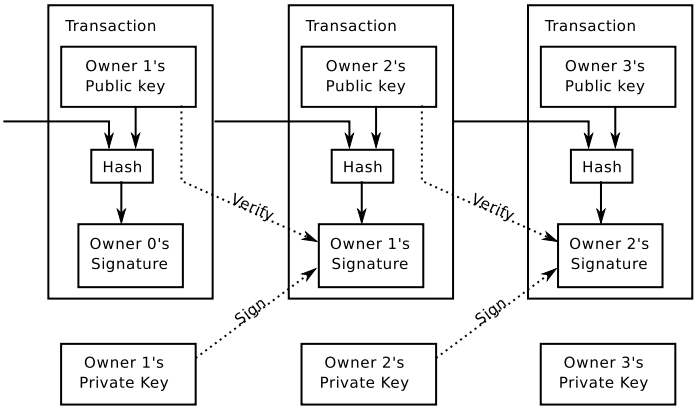

Figure 4.1: Three traditional Namecoin transactions.

initial genesis block, all blocks of data link to one another by referencing the hash of a previous block. Blocks are generated and appended to the chain at a relatively fixed rate by a process known as mining. The mining process is the solving of a proof-of-work (PoW) problem in a scheme similar to Adam Back's Hashcash: find a nonce that when passed through two rounds of SHA256 (SHA256^2) produces a value less than or equal to a target T. This requires a party to perform on average $\frac{1}{Pr[H \leq T]} = \frac{2^{256}}{T}$ amount of computations, but it is easy to verify afterwards that $\text{SHA256}^2(\text{msg}||n) \leq T$. Once the PoW is complete, the block (containing the back-reference hash, a list of transactions, and the PoW variables) is broadcasting to rest of the network, thus forming an append-only chain whose validity everyone can confirm. As each block is generated, the miner receives fresh Namecoins, thus introducing newly-minted Namecoins into the system at a fixed rate. Blocks cannot be modified retrospectively without requiring regeneration of the PoW of that block and all subsequent blocks, so the PoW locks blocks in the chain together. Nodes in the Namecoin network collectively agree to use the blockchain with the highest accumulation of computational effort, so an adversary seeking to modify the structure would need to recompute the proof-of-work for all previous blocks as well as out-perform the network, which is currently considered infeasible. [25]

Each block in the blockchain consists of a header and a payload. The header contains a hash of the previous block's header, the root hash of the Merkle tree built from the

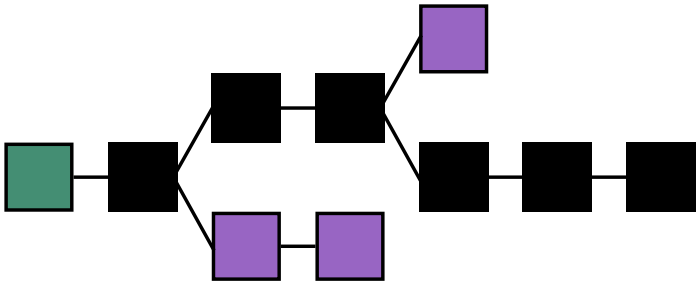

Figure 4.2: A sample blockchain.

transactions in this block, a timestamp, a target T, and a nonce. The block payload consists of a list of transactions. The root node of the Merkle tree ensures the integrity of the transaction vector: verifying that a given transaction is contained in the tree takes $log(n)$ hashes, and a Merkle tree can be built $n * log(n)$ time, ensuring that all transactions are accounted for. The hash of the previous block in the header ensures that blocks are ordered chronologically, and the Merkle root hash ensures that the transactions contained in each block are order chronologically as well. The target T changes every 2016 blocks in response to the speed at which the proof-of-work is solved such that Bitcoin miners take two weeks to generate 2016 blocks, or one block every 10 minutes. The change in target ensures that the difficulty of the proof-of-work remains relatively constant as processing capabilities increase according to Moore's Law. In the event that multiple nodes solve the proof-of-work and generate a new block simultaneously, the forked block becomes orphaned, the transactions recycled, and the network converges to follow the blockchain with the longest path from the genesis node, thus the one with the most amount of PoW behind it. [25]

Namecoin is particularly noteworthy in that it was the first systems to prove Zooko's Triangle false in a practical sense. The blockchain public ledger is held by every node in the distributed network, and human-meaningful names can be owned and placed inside it. The rules of the Namecoin network tell all participants to ensure that there are no name duplicates, and anyone holding the blockchain can verify this property. Names cannot be inserted retroactively due to the PoW, so these properties are ensured. Thus, Namecoin achieves all three properties in Zooko's Triangle. Namecoin is the most commonly-used alternative DNS system and is noted for its security and censorship resistance. In 2014,

Namecoin was recognized by ICANN as the most well-known example of a PKI and DNS system with an emphasis of distributed control and privacy, a growing trend in light of the revelations about the US National Security Agency (NSA) by Edward Snowden. [26]

While Namecoin is perhaps the most well-known secure distributed DNS system, it too is not applicable to Tor's environment. One of the main problems is one of practicality: it is unreasonable to require all Tor users to be able to download the entire Namecoin blockchain, which currently stands at 2.05 GB as of January 2015, [27] in order to know DNS records and verify the uniqueness of names. While this burden could be shifted to Tor nodes, Tor clients must then be able to trust the nodes to return an accurate record or even the correct block that contained that record. If the client queried the Namecoin network for DNS information through a Tor circuit, they would have no way of verifying the accuracy of the information without holding a complete blockchain to verify it. Secondly, the hidden service operator would have to prove that he owned both the ECDSA private key attached to the Namecoin record and the private RSA key attached to his hidden service in a manner that is both publicaly confirmable and didn't compromise his identity or location. These problems make Namecoin difficult to integrate securely into Tor. While we recognize Namecoin for its novelty and OnionNS shares some similarities with Namecoin, Namecoin itself is not a viable solution here.

Namecoin meets only some of our initial requirements; anonymity during registration can be met when the hidden service operator uses a Tor circuit, anonymity or privacy-enhancement during lookup can be met when the Tor client uses a Tor circuit for the query, public confirmability of the name to the hidden service is not easily met, domain name uniqueness is met by the Namecoin network but not easily proven without access to the entire blockchain, the distributed property is met by the nature of the network, simplicity is not fully met by Namecoin and further software would have be developed to accomplish this objective for Tor integration, backwards compatibility is unknown but could theoretically be met by certain designs, and resistance to malicious modifications or falsifications is met by the network but again not easily proven without access to the entire blockchain.

CHAPTER 5

SOLUTION

5.1 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.

5.2 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 5.5 and how

they are used throughout sections 5.3 and 5.6.

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 Internet 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 5.1. Domain names point to *destinations* – other domain names with either the .tor or .onion TLD.

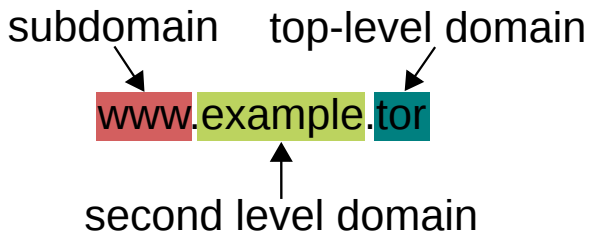

Figure 5.1: A sample domain name: a sequence of labels separated by delimiters. In OnioNS, hidden service operators build associations between second-level domain names and their hidden service address.

The Internet 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 5.5.1 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 5.6.1 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 5.6.3.

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 5.6.2.

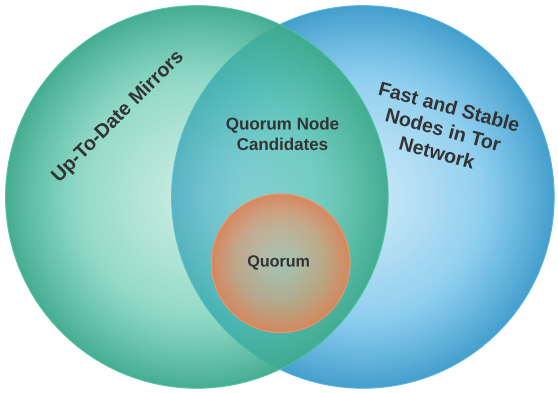

Figure 5.2: The relationship between Mirrors, Quorum Candidates, and the Quorum. A Mirror is any machine that holds the OnioNS Pagechain, Quorum Candidates are both up-to-date Mirrors and reliable Tor nodes, and the Quorum is randomly selected from the pool of Quorum Candidates.

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.

5.3 Basic Design

OnioNS is build on a fundamental sequence of operations, as illustrated in Figure 5.3:

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 5.6.1, 5.6.2, and 5.6.3, 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.

5.3.1 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

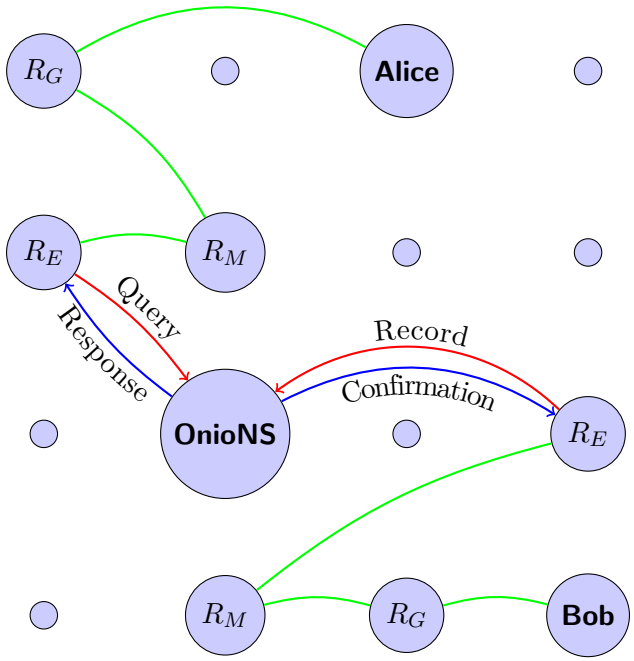

Figure 5.3: Bob uses a Tor circuit (guard, middle, and exit Tor routers) to anonymously broadcast a record to OnioNS. Alice uses her own Tor circuit to query the system for a domain name, and she is given Bob's record in response. Then Alice connects to Bob by Tor's hidden service protocol (section 1.2.4).

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 5.4.

5.3.2 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.

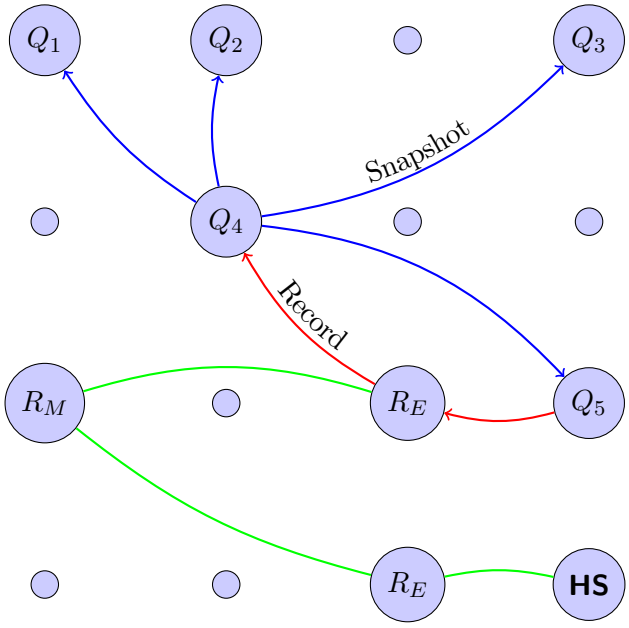

Figure 5.4: Bob uses his existing circuit (green) to inform Quorum node Q_4 of the new record. Q_4 then floods it via Snapshots to all other Quorum nodes. Each node stores it in their own Page for long-term storage. Bob confirms from another Quorum node Q_5 that his Record has been received.

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 5.6.2. Thus the Pagechain is at least partially self-healing. We illustrate this in Figure 5.5.

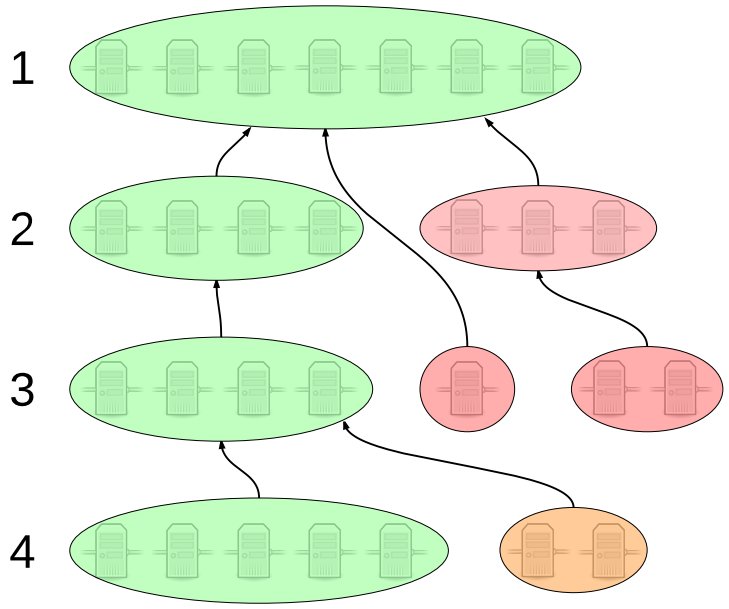

Figure 5.5: An example Pagechain across four Quorums. Each Page contains a references to a previous Page, forming a distributed append-only chain. Quorum_1 is honest and maintains reliable flooding communication, and thus has identical Pages. Quorum_2's largest cluster is likewise in agreement, but there 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.

5.3.3 Client Request

Let Bob and Dave be two hidden services, and assume that OnioNS knows a Record from Bob containing an "example.tor → example0uyw6wgve.onion" association, and a Record from Dave 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 Internet 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 Dave'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 Internet, 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 Dave-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 5.5.1) so this algorithm always completes. We discuss minor optimizations and security enhancements to this procedure in section 5.6.3.

5.4 Primitives

5.4.1 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. [28] 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. [29] 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. [30] [31] 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. [32]

5.4.2 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.

5.5 Data Structures

5.5.1 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 5.1: 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.

5.5.2 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.

5.5.3 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.

5.6 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.

5.6.1 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 5.6.

```
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 5.6: 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 5.6.3.

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.

5.6.2 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 5.7: 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 5.8: A sample empty Snapshot.

2. Charlie obtains the consensus document *cd* issued on day $\left\lfloor \frac{q}{\Delta q} \right\rfloor$ 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 checks that *prevHash* references some page from the previous Quorum.

   (b) Charlie calculates $h = H(\textit{prevHash} \parallel \textit{recordList} \parallel \textit{consensusDocHash})$.

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

   (d) 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 $\textit{consensusDocHash} = H(cd)$.

   (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. [10]

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 [10]: 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 5.9.

   (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 5.9: 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 5.10.

   (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.

5.6.3 Tor Clients

Let Alice be a Tor client. We assume now that Alice has chosen Charlie as her domain resolver and that Charlie is not a member of the current Quorum.

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.

```
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 5.10: A Page containing an example Record. This Page is the result of the Snapshot in Figure 5.9 into an empty Page.

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

4. Alice uses f to randomly scramble qc.

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

Domain Query

There are three verification levels in a Domain Query, each providing progressively more

verification to Alice that the Record she receives is authentic, unique, and trustworthy. At verification 0, the default:

1. Alice constructs a Tor circuit to Charlie, so from Charlie's perspective she is anonymous.
2. Alice provides a .tor domain name d into the Tor Browser. The Internet has no .tor TLD, so d is not processed as a traditional domain name would be.
3. If d's highest-level name is "www", Alice removes that name.
4. Alice asks Charlie for the most recent Record r associated with d at verification level 0.
5. Charlie reviews his Pagechain reverse-chronological order until he finds a Record containing d, which he returns to Alice.
6. Alice validates r via the Record Validation protocol. If it does not validate, she changes to another resolver and repeats this protocol.
7. If the destination for d in r uses a .tor TLD, d becomes that destination and Alice jumps back to step 3.
8. Otherwise, the destination must have a .onion TLD, which Alice looks up by the Tor hidden service protocol.
9. Alice checks that r's *pubHSKey* matches the one in Tor's distributed hash table.
10. Alice sends the original d to the hidden service.

Verification level 1 proceeds nearly identical to level 0. Alice sends "level 1" to Charlie in step 4 and Charlie returns the Page p containing r. Then Alice can check the integrity of p via the Page Selection protocol and can verify its authenticity via checking that $V_{ed}(m, C)$ returns true when m is the Page hash h and C is *fingerprint*'s public Curve25519 NTor key.

At Verification level 2, Alice sends "level 2" to Charlie in step 4 and Charlie returns p from level 1 and all digital signatures on h from all Quorum nodes that agreed on p. Then Alice can perform width verification: she can see that a large percentage of Quorum nodes maintained p, thus increasing the trustworthiness of both p and r.

Alice can be certain that the Record r she receives is authentic and that it contains a domain name d that is unique by performing a full Synchronization and obtaining Charlie's Pagechain for herself, but this is impractical in most environments. It cannot be safely assumed that Alice has storage capacity to hold all the Pages in the Pagechain. Additionally, Tor's median circuit speed is often less than 1 MiB/s, [10] so for convenience data transfer must be minimized. Therefore Alice can simply fetch minimal information and rely on her trust of Charlie and the Quorum.

Onion Query

Alice may also issue a reverse-hostname lookup to Charlie to find second-level domains that resolve to a given .onion address. This request is known as an Onion Query. Charlie performs a reverse-chronological search in the Pagechain for Records that .onion as a destination and returns the results corresponding to the verification level. Onion Queries have a high likelihood of failure as not every .onion name has a corresponding .tor domain name, but all OnioNS domain names will have Forward-Confirmed Reverse DNS match.

5.7 Optimizations

There are several improvements that be made upon the basic design protocols that significantly enhance the performance of Mirrors when responding to Domain or Onion Requests. We also introduce the Hashtable Bitset data structure, which improves security on the client end.

5.7.1 AVL Tree

An AVL tree is a self-balancing binary search tree with $\mathcal{O}(\log(n))$ time for search, insert, and delete. OnioNS mirrors can cache all the Records in its local Pagechain in an

AVL tree. After the Pagechain is validated, the Mirror iterates through the Pagechain in chronological order and constructs a list of associations between a second-level domain name and the location in the Pagechain of the Record containing that domain. The Mirror then inserts that list into the AVL tree, where sorting occurs by alphabetical comparison of domain names. This effectively transforms the lookup time of Records for Domain Queries from $\mathcal{O}(n)$ to $\mathcal{O}(\log(n))$ in the worst case.

As the Mirror handles new Records sent to Quorum nodes, it must update this AVL tree. Create Records trigger an insert operation, Delete Records cause a deletion, and all other Records update a location pointer to the more recent Record.

5.7.2 Trie

We also suggest utilizing a trie, a digital tree, for efficiently structuring .onion addresses and optimizing Onion Query lookups. If each node in the trie is a character in the .onion address, the trie has a branching factor of 58 and a maximum depth of 16. Let the leaves of the trie be the location of the most recent Record in the Pagechain that has that address as a destination. Like the AVL tree, Mirrors must take care to update the trie cache when processing new Records, but this is efficient as trie search, insertion, and deletion all occur in $\mathcal{O}(1)$.

5.7.3 Hashtable Bitset

A hashtable bitset is a special and highly compact adaptation of a traditional hashtable. We introduce a hashtable bitset for two purposes: 1) to prove the non-existence of a domain name, and 2) to improve the efficiency of lookups for non-existent domain names from $\mathcal{O}(\log(n))$ through the AVL tree to $\mathcal{O}(1)$ time. This first property is a challenge often overlooked in other domain name systems: even if domain names can be authenticated by a client (e.g. OnioNS Records or SSL certificates) a DNS resolver may lie about the non-existence and claim a false negative. In OnioNS, Alice can download the entire Pagechain and confirm for herself, but as we stated earlier this is not practical. Alice could also query another trusted source such as Quorum Candidates, but this approach does not scale well.

Instead, a trusted authority can sign a hashtable bitset once and allow Mirrors to prove non-existence for any domain name.

As an extension of an ordinary hashtable, the hashtable bitset maps keys to buckets, but here we only track the existence of a key but not the keys themselves. Therefore each bucket in the structure is represented as a bit, creating a compact $z * n$-length bitset, where z is a scaling factor. Let the hashtable use the $H(m)$ function from section 5.4.1. A Mirror constructs a hashtable bitset in $\mathcal{O}(n)$ time by the following algorithm:

1. Construct the hashtable bitset hb with all bits set to zero.
2. Construct an empty AVL tree t.
3. For each domain name d in each Record r in the Pagechain,
   (a) If $hb(H(d))$ is 1, add $H(d) \rightarrow r$ to hb.
   (b) If $hb(H(d))$ is 0, set $hb(H(d))$ to 1.
4. Divide hb into k equally-sized section and digitally sign each section with $S_{ed}(m, c)$.
5. Digitally sign t with $S_{ed}(m, c)$.
6. Make hb, t, and their signatures for download.

A Mirror, Charlie, then obtains the signatures from a Quorum node. When Alice requests a domain name d that does not exist,

1. Charlie returns back a 404 message, the relevant section of hb, and the signature on that section.
2. Alice verifies the signature on the hb section.
3. Alice checks that $hb(H(d))$ is a 0. If it is, she knows the domain name does not exist.
4. If $hb(H(d))$ is a 1, Alice asks Charlie for t.
5. Charlie returns t and the signature on t.

6. Alice verifies the signature on t.

7. Alice confirms that $H(d)$ does not exist within t, demonstrating that the domain 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 $k = 1$.

CHAPTER 6

ANALYSIS

6.1 Security

Now we examine and compare OnioNS' central protocols against our security assumptions and expected threat model.

6.1.1 Quorum Selection

The Quorum nodes have greater attack capabilities than any other class of participants in OnioNS. They have active responsibility over the front of the Pagechain and they must receive, flood, and process new Records from hidden service operators. Malicious Quorums may ignore or substitute received Records and or may attempt to mislead Mirrors. In our threat model, we assume that an attacker, Eve, already has control of some fixed number f_E of routers in the Tor network, and that her nodes may maliciously collude. We also assume that Eve does does not have motivation to compromise Tor in response to the presence of OnioNS and that she cannot predict future Quorums. We therefore feel it safe to examine our Quorum protocols and explore the likelihood of attacks within a probabilistic environment.

The Quorum Derivation protocol selects an L_Q-sized subset of routers from the set of Quorum Candidates, and rotates this selection every Δq days. The optimal selection of L_Q and Δq is dependent on both security and performance analysis; our security analysis introduces a bound on both L_Q and Δq. For the following evaluations, we feel it safe to discard threats that have probabilities at or below $\frac{1}{2^{128}} \approx 10^{-38.532}$ — the probability of Eve randomly guessing a 128-bit AES key, a threat that would violate our security assumptions.

Our security analysis assumes that L_Q will be selected from a pool of 5,400 Quorum

Candidates — the number, as of April 2015, of Tor routers with the Fast and Stable flags, whom we assume are all up-to-date Mirrors. Let L_E be the number of Quorum nodes under Eve's control. Then Eve controls the Quorum if the L_E routers become the largest agreeing subset in the Quorum, which can occur if either more than $\frac{L_Q - L_E}{2}$ honest Quorum nodes disagree or if $L_E > \frac{L_Q}{2}$. Here we examine the second scenario, which can be theoretically modelled.

Quorum selection is mathematically an L_Q-sized random sample taken from an N-sized population without replacement, where the population contains a subset of f_E entities that are considered special. Then the probability that Eve controls k Tor routers in the Quorum is given by the hypergeometric distribution, whose probability mass function (PMF) is $\frac{\binom{f_E}{k}\binom{N-f_E}{L_Q-k}}{\binom{N}{L_Q}}$. Then the probability that $L_E > \frac{L_Q}{2}$ is given by $\sum_{x=\lceil \frac{L_Q}{2} \rceil}^{L_Q} \frac{\binom{f_E}{k}\binom{N-f_E}{L_Q-k}}{\binom{N}{L_Q}}$. Odd choices for L_Q prevents the possibility of network disruption when the Quorum is evenly split in terms of the current Page. We examine the probability of Eve's success for increasing amounts of f_E in Figure 6.1.

Figure 6.1 shows that a choice of $L_Q = 31$ is suboptimal: the probabilities are above the $10^{-38.532}$ threshold for even small levels of collusion. $L_Q = 63$ likewise fails for percentages above approximately 2, though choices of 127, 255, and 511 fail at levels above approximately 8, 16, and 25 percent, respectively. The figure also suggests that larger Quorums are superior with respect to security. Small Quorums are also less resilient to DDOS attacks at the Quorum in general.

If we assume that Eve controls 10 percent of the Tor network, then we can examine the impact of the longevities of Quorums; over a fixed period of time, slower rotations suggests a lower cumulative chance of selecting any malicious Quorum. If w is Eve's chance of compromise, then her cumulative chances of compromising any Quorum is given by $1-(1-2)^t$. This gives us a bound estimate on Δq. We estimate this over 10 years in Figure 6.2.

Figure 6.2 suggests that while slow rotations (i.e a period of 7 days) generates orders of magnitude less chance than fast rotations, the choice of L_Q is far more significant. Like

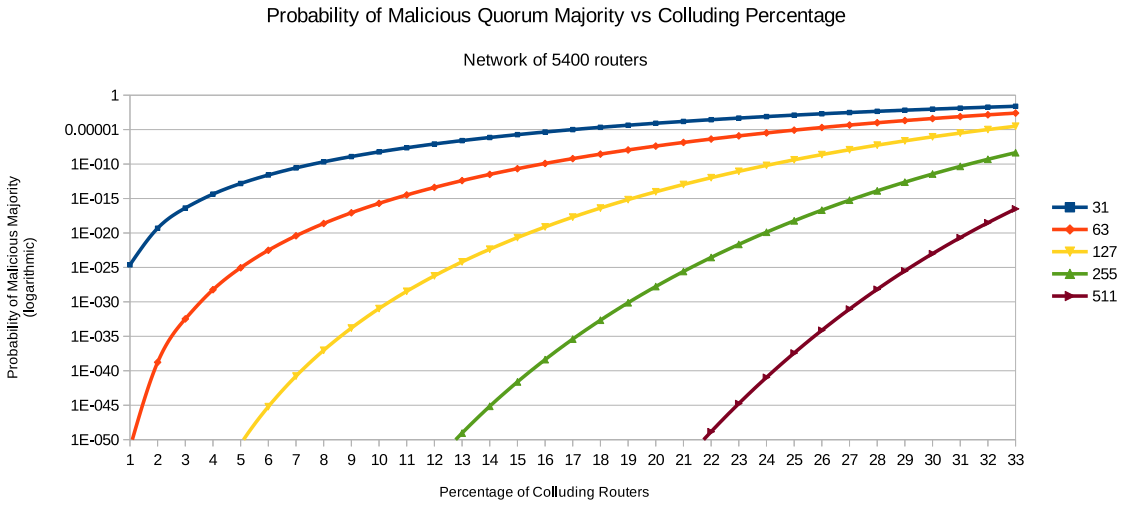

Figure 6.1: The probability that Eve controls the majority of the Quorum is given by the PMF of the hypergeometric distribution. We fix N at 5,400 nodes and graph Eve's success probability as a function of an increasing percentage of Eve-controlled colluding routers. We examine five selections for L_Q: 31, 63, 127, 255, and 511. We do not consider percentages beyond 33 percent as 33 percent represents a complete compromise of the Tor network: it is near 100 percent that the three routers selected during circuit construction are under Eve's control, a violation of our security assumptions.

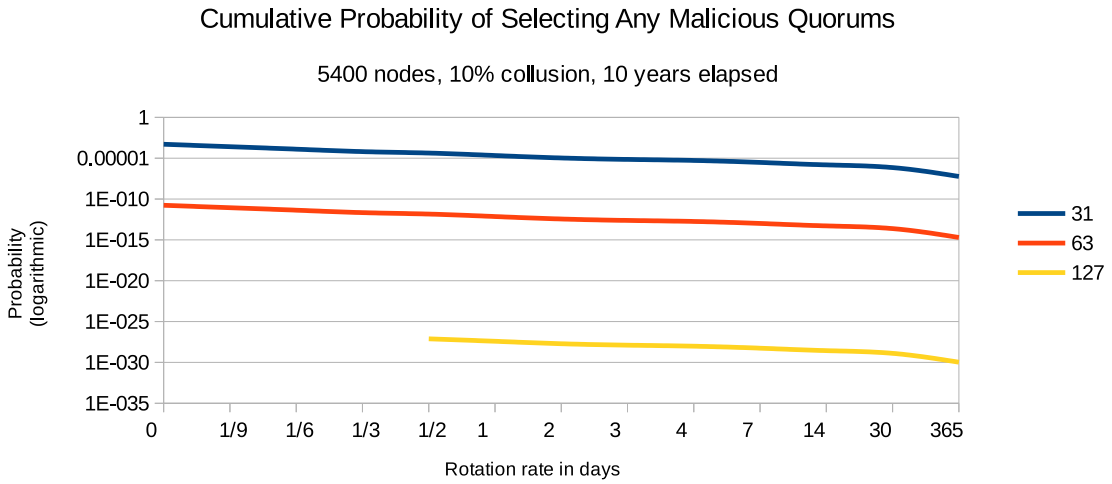

Figure 6.2: The cumulative probability that Eve controls any Quorum at different rotation rates. We assume 10 percent collusion in a network of 5400 Tor routers, and view across 10 years. We do not graph L_Q values of 255 or 511 as they generate probabilities far below our $10^{-38.532}$ threshold; $L_Q = 255$ and $L_Q = 511$ produce values less than 10^{-58} and 10^{-134}, respectively.

Figure 6.1, it also shows that $L_Q = 31$ and $L_Q = 61$ are relatively poor choices.

If a selected Quorum is malicious, fast rotation rates will minimize the duration of any disruptions, as shown in Figure 6.3. This figure suggests that fast rotations are optimal in that respect, a contradiction to Figure 6.1. However, given the very low statistical likelihood of selecting a malicious Quorum, we consider this a minor contribution to the decision.

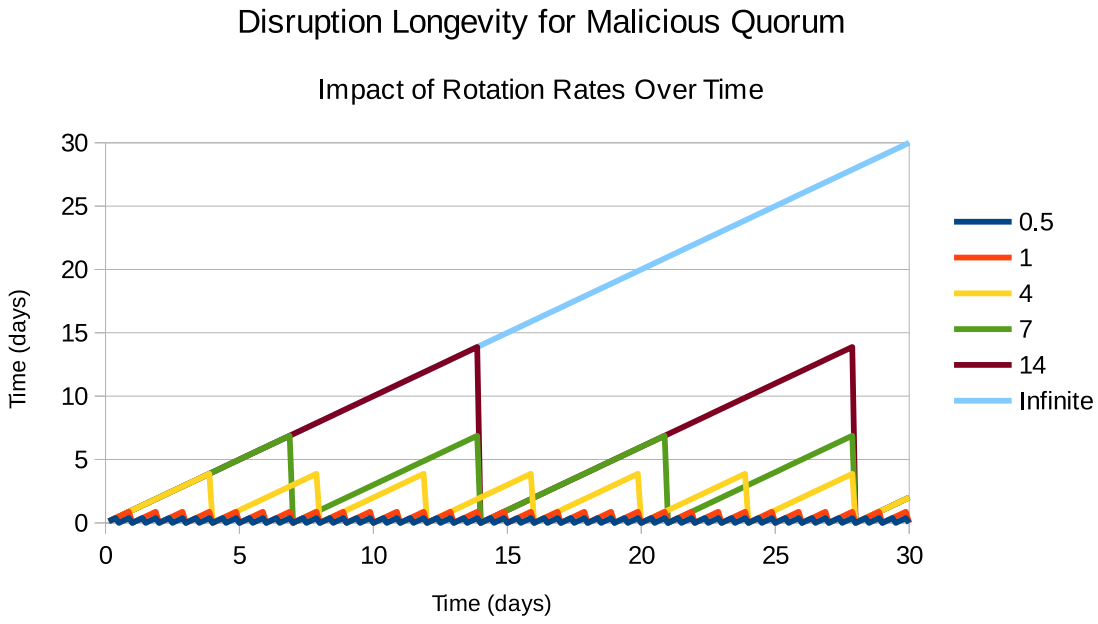

Figure 6.3: The duration of malicious Quorums as a function of different rotation rates. Quorums that have very short livetimes (and are thus rotated quickly) minimize the duration of any malicious activity.

Although a malicious Quorum would have the capabilities to deploy a variety of attacks on the network, the proper selections of $L_Q \geq 127$ and $\Delta q \geq 1$ reduces the likelihood of this occurring to near-zero probabilities. We consider this a stronger solution than introducing countermeasures to those attacks. Based on our security analysis, we suggest $L_Q \geq 127$ and $\Delta q \geq 1$.

6.1.2 Entropy of Tor Consensus Documents

OnioNS can be adapted and deployed on any network that is both fully connected

and that contains a common source of entropy. In Tor's case, we use the Tor consensus documents to derive the Quorum, but the consensus documents were designed as a common source of *information*, and thus we must prove that the consensus documents contain enough entropy to be securely used. If there is not enough entropy, Eve can subvert the Quorum Derivation protocol in a variety of attack vectors, including to include her malicious routers in the Quorum, or to reject specific honest routers. As before, we also conclude that ensuring sufficient entropy in these documents is the more effective solution to these threats.

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, however they are difficulty and slow to carry out in

practice as as Tor authority nodes grant consensus weight to new Tor nodes very slowly. 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, large-scale Sybil attacks are financially demanding and time-consuming to Eve.

Initial results suggest that the Levenshtein distance between consensus documents on consecutive days is large, but that's not the same thing as entropy.

The entropy of the consensus documents can be easily further increased if each of the nine authority nodes contribute a significant amount of random characters into the consensus documents.

6.1.3 Outsourcing Record Proof-of-Work

The Record Generation protocol can safely take place within an offline machine under the operator's control. We designed the protocol to require resigning the Record fields after every round of scrypt so as to require the owner of the private key to also perform the scrypt proof-of-work. Scrypt introduces a cost and its memory demands increases the difficulty of custom hardware and massively-parallel attacks, effectively limiting adversaries to the same hardware as legitimate users.

However, our protocol does not entirely prevent a hidden service operator from outsourcing the computation to secondary resource. Let Bob be the hidden service and Craig the owner of the computational resource. We assume that Craig does not have Bob's private key. Then,

1. Bob creates an initial Record R and completes the *type*, *nameList*, *contact*, *timestamp*, and *consensusHash* fields.

2. Bob sends R to Craig.

3. Let *central* be *type* $\|$ *nameList* $\|$ *contact* $\|$ *timestamp* $\|$ *consensusHash* $\|$ *nonce*.

4. Craig generates a random integer K and then for each iteration j from 0 to K,

   (a) Craig increments *nonce*.

(b) Craig sets PoW as PoW($central$).

(c) Craig saves the new R as C_j.

5. Craig sends all $C_{0 \leq j \leq K}$ to Bob.

6. For each Record $C_{0 \leq j \leq K}$ Bob computes

(a) Bob sets *pubHSKey* to his public RSA key.

(b) Bob sets *recordSig* to $S_d(m, r)$ where $m = central \parallel pow$ and r is Bob's private RSA key.

(c) Bob has found a valid record if $H(central \parallel pow \parallel recordSig) \leq 2^{difficulty * count}$

Our protocol ensures that Craig must always compute more scrypt iterations than necessary; Craig cannot generate *recordSig* and thus cannot compute if the hash is below the threshold. Moreover, the scrypt work incurs a cost onto Craig that must be compensated financially by Bob. Thus the Record Generation protocol places a lower bound on the cost paid by Bob.

6.1.4 Sybil Attack

What could a Mirror lie about, and what are the implications? I'm really not sure what could happen here because of the Page signatures.

One countermeasure against this attack is SSL certificates: if a

Although Tor hosts approxiemetly 25,000 hidden services, as of April 2015 only three use browser-trusted SSL certificates: Blockchain.info at https://blockchainbdgpzk.onion, Facebook at https://facebookcorewwwi.onion, and The Intercept's SecureDrop instance at https://y6xjgkgwj47us5ca.onion. Modern web browsers check that the SSL certificate matches the user-entered domain name, so in this application a Certificate Authority must sign the .tor domain, rather than the .onion address. However, given the low degree of practicality of this attack and the overhead of a redundant layer of TLS encryption, we do not recommend applying SSL certificates in the general sense.

6.1.5 False Negatives Claims

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.

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.

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.

6.2 DNS Leakage

leaking onto the clearnet DNS, papers on this

6.3 Reliability

6.4 Objectives Assessment

OnioNS achieves all of our original requirements:

1. **The system must support anonymous registrations** — OnioNS Records do

not contain any personal or location information. The PGP key field is optional and may be provided if the hidden service operator wishes to allow others to contact him. However, the operator may be using an email address and a Web of Trust disassociated from his real identity, in which case no identifiable information is exposed.

2. **The system must support privacy-enhanced lookups** — OnioNS performs Domain and Onion Queries through Tor circuits, and under our original assumption that circuits provide strong guarentees of client privacy and anonymity, resolvers cannot sufficiently distinguish users to track their lookups.

3. **Clients must be able to authenticate registrations** — OnioNS Records are self-signed, enabling Tor clients to verify the digital signature on the domain names and check the public key against the server's key during the hidden service protocol. This ensures that the association has not been modified in transit and that the domain name is authentic relative to the authenticity of the destination server.

4. **Domain names must be provably or have a near-certain chance of being unique** — Tor hidden services .onion addresses are cryptographically generated with a key-space of $58^{16} \approx 2^{93.727695922}$ and domain names within OnioNS are provably unique by anyone holding a complete copy of the Pagechain.

5. **The system must be distributed** — The responsibilities of OnioNS are spread out across many nodes in the Tor network, decreasing the load and attack potential for any single node. The Pagechain is likewise distributed and locally-checked by all Mirrors, and although its head is managed by the Quorum, these authoritative nodes have temporary lifetimes and are randomly selected. Moreover, Quorum nodes do not answer queries, so they have limited power.

6. **The system must be simple and relatively easy to use** — Domain Queries are automatically resolved and require no input by the user. From the user's perspective, they are taken directly from a meaningful domain name to a hidden service. Users no

longer have to use unwieldy .onion addresses or review third-party directories, OnioNS introduces memorability to hidden service domains.

7. **The system must be backwards compatible with existing protocols** — OnioNS does not require any changes to the hidden service protocol and existing .onion addresses remain fully functional. Our only significant change to Tor's infrastructure is the mechanism for distributing hashes for the Quorum Qualification protocol, but our initial technique for using the Contact field minimizes any impact. We also hook into Tor's TLD checks, but this change is very minor. Our reference implementation is provided as a software package separate from Tor per the Unix convention.

Finally, we meet our optional performance objectives:

1. **The system should not require clients to download the entire database** — Only Mirrors hold the Pagechain, and clients do not need to obtain it themselves to issue a Domain Query. Therefore clients rely on their existing and well-established trust of Tor routers when resolving domain names. However, clients may optionally obtain the Pagechain and post Domain Queries to localhost for greater privacy and security guarantees.

2. **The system should not introduce significant burdens to the clients** — Record verification should occur in sub-second constant time in most environments, and Ed25519 achieves very fast signature confirmation so verifying Page signatures at level 1+ takes trivial time. However, clients also verify the Record's proof-of-work, so for some scrypt parameters the client may spend non-trivial CPU time and RAM usage confirming the one scrypt iteration required to check. We must therefore choose our parameters carefully to reduce this burden especially on low-end hardware.

3. **The system should have low latency** — Domain Queries without any packet delays over Tor low-latency circuits. Its exact performance is largely dependent on circuit speed and the client's verification speed.

We therefore believe that we have squared Zooko's Triangle; OnioNS is distributed, enables hidden service operators to select human-meaningful domain names, and domain names are guaranteed unique by all participants.

CHAPTER 7

IMPLEMENTATION

7.1 Prototype Design

We have developed a prototype of OnioNS and have implemented most of the hidden service, server, and client protocols in C++11. We package the software on the Launchpad online build system and support Linux Debian and derivates such as Ubuntu and Linux Mint. We use the Botan library for cryptographic operations, with the Mersenne Twister provided through the C++ Standard Template Library (STL). We have made our software online at github.com/Jesse-V/OnioNS. Our reference implementation uses the libjsoncpp header-only library for encoding and decoding purposes. The library is also available as the libjsoncpp-dev package in the Debian, Ubuntu, and Linux Mint repositories.

7.2 Experimentation

Todo: I will carry out experiments in test deployments of the Tor network and see what the demand is. I anticipate it being relatively lightweight. The proof-of-work will almost certainly be the computational bottleneck.

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

7.2.1 Proof-of-work

What are the optimal parameters for scrypt? What are the implications of setting it too high or too low?

How much bandwidth and time does this take?

How does the bandwidth and CPU load scale in response to a larger Quorum? Assuming reasonable clockskew, how off will the exchanges be? Are there any race conditions that need to be resolved?

Queres: How long does this take, and how can we improve efficiency? Can clients on even low-end hardware calculate the proof-of-work?

7.3 Results

This will be expanded/rewritten once I finish implementation and deploy it in test/prototyping networks. I plan on using Chutney.

CHAPTER 8

FUTURE WORK

OnioNS is designed as a plugin for Tor and to operate on top of Tor's hidden service protocol. While our implementation functions with a centralized solution, we will pursue its expansion onto larger and more realistic simulation environments until it is fully ready to integrate with Tor. To do this, we plan to develop and deploy a distributed edition of OnioNS on simulated Tor networks using the Chutney software, extend it to PlanetLab, before developing a Tor proposal document and merging it into the Tor source once it's approved by the Tor community.

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.

web of trust for .onion keys to thwart Sybil attack

Some open problems that I need to address include:

1. How frequently should domains expire? Are there any security risk in sending a Renew request?

2. How should unreachable or temporarily down nodes be handled? I'd like to know the percentage of the Tor network that is reachable at any given time.

3. How many nodes in the Tor network should be assumed to be actively malicious? What are the implications of increasing this percentage?

4. What attack vectors are there from the committee nodes, and how can I thwart the attacks?

5. What are the implications of a node intentionally voting the opposite way, or ignoring the request altogether?

6. What would happen if a node was too slow or did not have enough storage space to do its job properly?

7. What other open problems are there?

8. What related works are there in the literature that relate to the concepts I have created here?

CHAPTER 9

CONCLUSION

The security analysis strongly suggests that larger Quorums, such as 255 or 511, is superior choice. Figure ? suggests that even larger Quorums would be ideal. Furthermore, Figure ? suggests that a slow rotation with a large Quorum is ideal. However, Figure ? suggests that a fast rotation reduces the impact that a malicious Quorum could have on the system. Our performance analysis shows that a small Quorum reduces the load on all parties. We discard Quorum sizes of 512 and above: we consider the load too significant, and discard 31 and smaller: Figures ? and ? suggest that these Quorums are too small, and Figures ? and ? show that 63 is similar to 31, so we discard that size as well. Between the choices of 127 and 255, we select 127 as the suggested Quorum size and reflect this in our reference implementation. Our stability analysis and Figure ? suggests 7 or 14 for rotation, we choose 7.

We have introduced OnioNS, a Tor-powered distributed DNS that maps custom .tor domain names to traditional Tor .onion addresses. It enables hidden service operators to select a human-meaningful domain name and provide access to their service through that domain. We preserve the privacy and anonymity of both parties during registration, maintenance, and lookup, and furthermore allow Tor clients to verify the authenticity of domain names. Moreover, we rely heavily upon existing Tor infrastructure, which simplifies our design assumptions and narrows our threat model largely to attack vector already well-understood throughout the Tor literature.

The Pagechain is our novel distributed data structure which we introduce to resolve the difficulty of keeping large number of separate machines synchronized to a master database. While some other prominent distributed DNS schemes such as Namecoin use a blockchain, our Pagechain does utilize mining, but rather digital signatures which are already common

throughout the Tor network. The Pagechain is also partially self-healing in the face of corruption or malicious modification, and each involved participant can verify the integrity of their local Pagechain themselves. OnioNS' Pagechain also has a fixed maximal length, which places an upper bound on the networking, computational, and storage requirements for all participants, a valuable efficiency gain especially noticeable long-term.

OnioNS achieves all three properties of Zooko's Triangle: it is distributed, allows hidden service operators to select meaningful domain names, and all parties can confirm for themselves the uniqueness of domain names in the database. We provide a reference implementation in C++ that should enable Tor developers to deploy OnioNS into the Tor network with minimal effort. We believe that OnioNS will be a useful abstraction layer that will significantly enhance the usability and the popularity of Tor hidden services.

REFERENCES

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

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

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

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

[14] 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.

[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] T. T. Project, "Collector," https://collector.torproject.org/, 2015, accessed 16-Apr-2015.

[19] ——, "Tor directory protocol, version 3," https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt, 2015, accessed 16-Apr-2015.

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

[21] M. S. Ferdous, A. Jøsang, K. Singh, and R. Borgaonkar, "Security usability of petname systems," in *Identity and Privacy in the Internet Age*. Springer, 2009, pp. 44–59.

[22] M. Stiegler, "Petname systems," *HP Laboratories, Mobile and Media Systems Laboratory, Palo Alto, Tech. Rep. HPL-2005-148*, 2005.

[23] katmagic, "Shallot," https://github.com/katmagic/Shallot, 2012, accessed 4-Feb-2015.

[24] N. Mathewson, "Next-generation hidden services in tor," https://gitweb.torproject.org/torspec.git/tree/proposals/224-rend-spec-ng.txt, 2013, accessed 4-Feb-2015.

[25] K. Okupski, "Bitcoin developer reference," 2014.

[26] T. I. C. for Assigned Names and Numbers, "Identifier technology innovation panel - draft report," https://www.icann.org/en/system/files/files/report-21feb14-en.pdf, 2014, accessed 4-Feb-2015.

[27] bitinfocharts.com, "Crypto-currencies statistics," https://bitinfocharts.com/, 2015, accessed 4-Feb-2015.

[28] 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.

[29] 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.

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

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

[32] 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.