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

ABSTRACT

The Onion Name System:

Tor-powered Distributed DNS

for Tor Hidden Services

by

Jesse Victors, Master of Science

Utah State University, 2015

Major Professor: Dr. Ming Li
Department: Computer Science

Tor hidden services are anonymous servers of unknown location and ownership who can be accessed through any Tor-enabled web browser. They have gained popularity over the years, but still suffer from major usability challenges due to their cryptographically-generated non-memorable addresses. In response to this difficulty, in this work we introduce the Onion Name System (OnioNS), a privacy-enhanced distributed DNS that allows users to reference a hidden service by a meaningful globally-unique verifiable domain name chosen by the hidden service operator. We introduce a new distributed self-healing public ledger and construct OnioNS as an optional backwards-compatible plugin for Tor on top of existing hidden service infrastructure. We simplify our design and threat model by embedding OnioNS within the Tor network and provide mechanisms for authenticated denial-of-existence with minimal networking costs. Our reference implementation demonstrates that OnioNS successfully addresses the major usability issue that has been with Tor hidden services since their introduction in 2002.

(74 pages)

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 have 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 nonexistence of domain names with minimal networking costs without introducing any central authority.

ACKNOWLEDGEMENTS

I would like to thank my major professor, Dr. Ming Li, for his continual guidance. His analysis, suggestions, and descriptions of possible attacks were instrumental in developing and solidifying this work. I would also like to extend thanks for the rest of my committee: Dr. Dan Watson and Dr. Nick Flann for their support.

I would also like to thank Tor developers Roger Dingledine, Yawning Angel, and Nick Mathewson for their assistance with Tor technical support, Sarbajit Mukherjee for his commentary, and the Tor community for their continued support of OnioNS.

CONTENTS

Page

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 [3]. 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 [4]. 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 [5] [6].

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 [7]. 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 [8] [9] 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 [6]. 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.

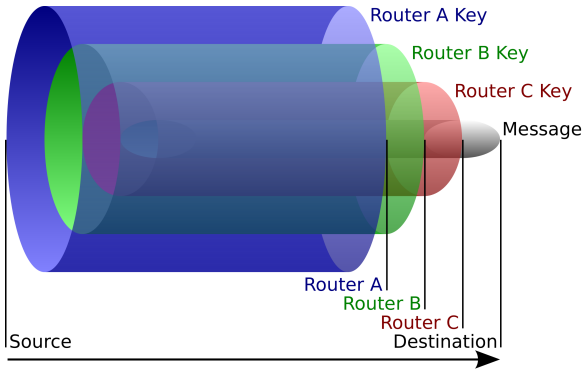

Figure 1.1: An example cell and message encryption in an onion routing scheme. Each router "peals" off its respective layer of encryption; the final router exposes the final destination.

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 [7] 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: [6] [7]

- **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. Each router remembers its respective key and can re-encrypts responses as it travels backwards up the circuit to the client, who then unwraps all the layers. 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 [6]. 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 [10] [11].

- **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 [6]. We describe the contents and format of this network information in section 1.2.2.

- **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 [7]. We provide additional details on the hidden service protocol in section 1.2.3 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 [1]. 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" [12] [13]. In 2014, BusinessWeek claimed that Tor was "perhaps the most effective means of defeating the online surveillance efforts of intelligence agencies around the world." [14]

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 [7].

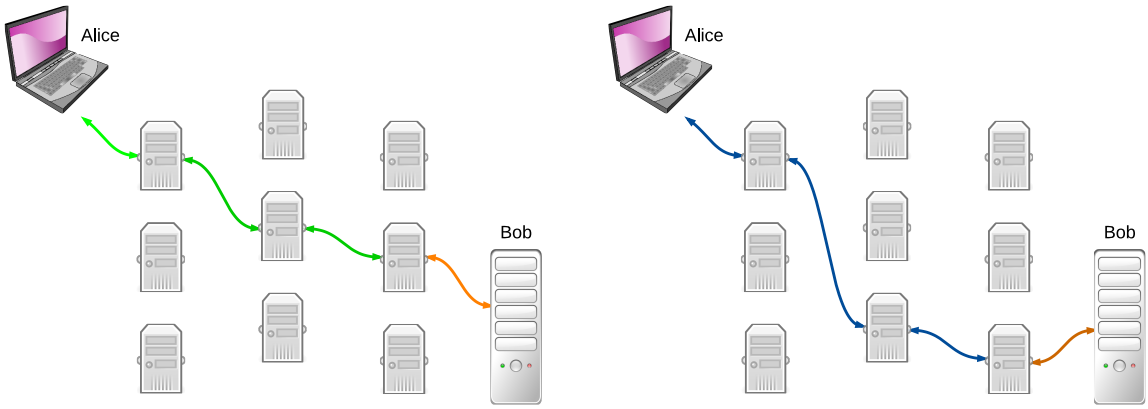

Figure 1.2: Alice communicates privately to Bob through a Tor circuit. Her communication path consists of three routers, an entry, middle, and exit. Although Bob's identity and location is known to Alice, the Tor circuit prevents Bob from knowing Alice's identity or location. At a later time, Alice may 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.

1.2.2 Consensus Documents

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 [15] [16]:

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 from the identity key of the above fields.

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.

- **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.
- **ntor-onion-key:** The router's public Curve25519 key.
- **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.

1.2.3 Hidden Services

Although Tor's primary and most popular use is for privacy-enhanced 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 [17].

Hidden services are known by their special domain name, which uses the .onion top-level domain (TLD). The Tor software considers this TLD a special case and does not attempt to resolve it on the Internet DNS, but rather through the Tor network. Hidden service addresses are algorithmically generated from the server's public RSA key; the address is the first 16 bytes of the base32-encoded SHA-1 hash of the server's RSA key. This builds

a publicly-confirmable one-to-one relationship between the public key and its address and allows hidden services to be referenced by their address in a distributed environment.

Let Bob be a hidden service. At startup, Bob randomly selects several router and builds Tor circuits to them. He then creates a hidden service descriptor, consisting of his public key B_K and a list of these routers. He signs the descriptor and sends a distributed hashtable within the Tor network, enabling the routers he chose to act as his *introduction points*. When Alice, a Tor client, obtains Bob's hidden service address though a backchannel, she queries this hashtable for Bob's address. Once she obtains Bob's hidden service descriptor, she then builds a circuit to one of the introduction points. Simultaneously, Alice also selects and builds a circuit to another relay, R_A. She encrypts R_A and a nonce with B_K and gives the result to R_A. Bob decrypts the message and builds a circuit to R_A (for security reasons, he uses the same guard router [10] [11]) and sends the nonce to Alice. Alice can then confirm Bob's authenticity and the two can begin communication over six Tor nodes: three established by Alice and three by Bob.

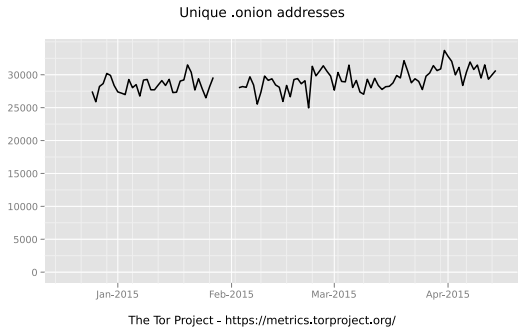

Figure 1.3: The number of unique .onion addresses seen in Tor's distributed hashtable between January through April 2015 [1] [2].

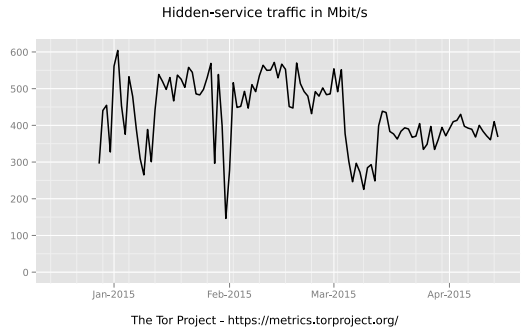

Figure 1.4: The amount of traffic generated by hidden services between January through April 2015 [1] [2].

1.3 Motivation

As hidden service addresses are algorithmically generated from the service's RSA key, there is a strong discontinuity between the address and the service's purpose. For example, a visitor cannot determine that `3g2upl4pq6kufc4m.onion` is the DuckDuckGo search engine without visiting the hidden service. Generally speaking, it is currently impossible to categorize or fully label hidden services in advance. Over time, third-party directories – both on the Clearnet and Darknet – have appeared in attempt to counteract this issue, but these directories must be constantly maintained and the approach is neither convenient nor does it scale well. Given the approximetly 25,000 hidden services on the Tor network, there is a strong need for a more complete solution to solve the usability issue.

1.4 Contributions

Our contribution to this problem is five-fold:

- We enable hidden service operators to construct a strong association between a unique human-meaningful domain name and their hidden service address.
- We described a distributed DNS database that is tamper-proof, self-healing, resistant to node compromise, and provides authenticated denial-of-existence.
- We provide OnioNS as a plugin for the existing Tor network, rather than introduce a new network. This simplifies our assumptions and largely reduces our threat model to attack vectors already well-understood on the Tor network.
- We enable Tor clients to verify the authenticity of a domain name against its corresponding hidden service address with minimal data transfers in a single query.
- We preserve the anonymity of both the hidden service and the privacy of Tor clients connecting to it.

To the best of our knowledge, this is the first alternative DNS for Tor hidden services which is distributed, secure, and usable at the same time.

CHAPTER 2

PROBLEM STATEMENT

2.1 Assumptions and Threat Model

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

1. We assume that Tor provides privacy and anonymity; if Alice constructs a three-hop Tor circuit to Bob with modern Tor circuit construction protocols and sends a message m to Bob, we assume that Bob can learn no more about Alice than the contents of m. This implies that if m does not contain identifiable information, Alice is anonymous from Bob's perspective, regardless of if m is exposed to an attacker, Eve. Identifiable information in m is outside of Tor's scope, but we do not introduce any protocols that cause this scenario.

2. We assume secure cryptographic primitives; namely that Eve cannot break standard cryptographic primitives such as AES, SHA-2, RSA, Curve25519, Ed25519, and the scrypt key derivation function. We assume that Eve maintains no backdoors or knows secret software breaks in the Botan or the OpenSSL implementations of these primitives.

3. We assume that not all Tor routers are honest; that Eve controls some percentage of Tor routers such that Eve's routers may actively collude. Routers may also be semi-honest; wiretapped but not capable of violating protocols. However, the percentage of dishonest and semi-honest routers is small enough to avoid violating our first assumption. We assume a fixed percentage of dishonest and semi-honest routers; namely that the percentage of routers under an Eve's control 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 while Tor traffic is purposely secret as it travels through the network, we consider OnioNS information public so we don't consider the inclusion of OnioNS a motivating factor to Eve.

4. If C is a Tor network status consensus, Q is an M-sized set randomly but deterministically selected from the Fast and Stable routers listed in C, and Q is under the influence of one or more adversaries, we assume that the largest subset of agreeing routers in Q are at least semi-honest.

2.2 Design Objectives

Here we enumerate a list of requirements that must be met by any DNS applicable to Tor hidden services. In Chapter 4 we analyse several existing prominent naming systems and show how these systems do not meet these requirements. In Chapters 5 and 6 we demonstrate how we overcome them with OnioNS.

1. **The system must support anonymous registrations.** The system should not require any personally-identifiable or location information from the registrant.

2. **The system must support privacy-enhanced queries.** Clients should be anonymous, indistinguishable, and unable to be tracked by name servers.

3. **Registrations must be authenticable.** Clients must be able to verify that the domain-address pairing that they receive from name servers is authentic relative to the authenticity of the hidden service.

4. **Domain names must be globally unique.** Any domain name of global scope must point to at most one server. For naming systems that generate names via cryptographic hashes, the key-space must be of sufficient length to resist cryptanalytic attack.

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 compromise the privacy of both users and hidden services or may not allow anonymous registrations.

6. **The system must be relatively easy to use.** It should be assumed that users are not security experts or have technical backgrounds. The system must resolve protocols with minimal input from the user and hide non-essential details.

7. **The system must be backwards compatible.** Naming systems for Tor must preserve the original Tor hidden service protocol, making the DNS optional but not required.

8. **The system should be lightweight.** In most realistic environments clients have neither the bandwidth nor storage capacity to hold the system's entire database, nor the capability of meeting significant computation burdens.

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 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 [18] [19].

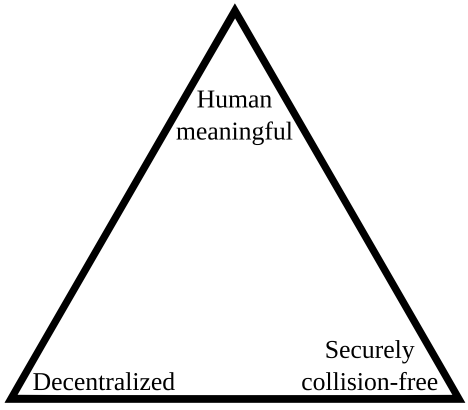

Figure 3.1: Zooko's Triangle.

Some examples of naming systems that achieve only two of these properties include:

- **Securely unique and human-meaningful** — Internet domain names are memorable and provably collision free, but the Internet DNS with a hierarchical structure with 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 addresses, 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.

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, the former is addressed by SSL certificates and a chain of trust to root Certificate Authorities, while the latter remains a possible attack vector. Of equal importance, however, is the capability to verify a claim of non-existence by a name server. This is a weakness often overlooked in other DNSs and resolving this problem is not easy. While DNSSEC does provide an extension for this purpose, DNSSEC has not seen widespread use and to our knowledge no alternative DNS provides mechanisms for authenticated denial-of-existence.

CHAPTER 4

EXISTING WORKS

We now examine several workarounds and existing naming systems against the design objectives listed in Section 2.2.

4.1 Address Manipulation

Although they are not separate naming systems in their own right, several systems have been proposed to directly improve the readability of hidden service addresses. These include vanity key generators and different encoding schemes.

Shallot is a vanity key generator which generates by brute-force many RSA keys in attempt to find one that has a desirable hash [20]. For example, a hidden service operator may wish to start his service's address with a meaningful noun so that others may more easily recognize it. Shallot has been used successfully for both common and high-profile hidden services, such as blockchainbdgpzk.onion, an official mirror of Blockchain.info; facebookcorewwwi.onion, Facebook's hidden service; and freepress3xxs3hk.onion, the Freedom of the Press's SecureDrop instance. However, Shallot is only partially successful at enhancing readability because the size of the domain key-space is too large to be fully brute-forced in any reasonable length of time [20]. This situation is expected to get worse over time as Tor plans to increase the length of hidden service addresses [21]. If the address key-space was reduced to allow a full brute-force, the system would fail to be collision-free.

Nicolussi suggested changing the address encoding from base32 to a delimited series of words, using a dictionary known in advance by all parties [17]. Like Shallot, Nicolussi's encoding is cosmetic and only partially improves the recognition and readability of an address but does nothing to alleviate the logistic problems of manually entering in the address into the Tor Browser.

4.2 Centralized or Zone-Based DNS

4.2.1 Internet DNS

The Internet DNS is another one candidate and is already well established as a fundamental abstraction layer for Internet routing. Despite its widespread use and extreme popularity, the Internet DNS suffers from several significant shortcomings and security issues that make it inappropriate for use by Tor hidden services. With the exception of extensions such as DNSSEC, the Internet DNS by default does not use any cryptographic primitives. DNSSEC is primarily designed to prevent forgeries and DNS cache poisoning from intermediary name servers and it does not provide any degree of query privacy [22]. Additional extensions and protocols such as DNSCurve [23] have been proposed, but DNSSEC and DNSCurve are optional and have not yet seen widespread full deployment across the Internet. Traditional DNS lookups may be intercepted and modified by MITM attacks, user privacy may be compromised by wiretapping DNS lookups, and the system is by default vulnerable to DNS cache poisoning.

The lack of default security in Internet DNS and the financial expenses involved with registering a new TLD casts significant doubt on the feasibility of using it for Tor hidden services. Furthermore the system meets only a few of our design requirements: although the system is easy to use, the system is hierarchical but not truly distributed, domain registrars typically require the owner to reveal significant amounts of identifiable information, registrations are not confirmable except through expensive SSL certificates issues by a central authority, and lookups occur by default without any privacy enhancements. These issues make the Internet DNS ill-suited for Tor hidden services.

4.2.2 GNU Name System

The GNU Name System [22] (GNS) is a zone-based alternative DNS. GNS describes a hierarchical zones of names (using the .gns pseudo-TLD) with each user managing their own zone and distributing zone access peer-to-peer within social circles. While GNS' design guarantees the uniqueness of names within each zone and users are capable of selecting

meaningful nicknames for themselves, GNU does not guarantee that names are *globally* unique. Furthermore, the selection of a trustworthy zone to use would be a significant challenge for using GNS for Tor hidden services and such a selection no longer makes the system distributed. However, GNS does meet many, but not all, of our requirements, so we consider GNS a very impressive system and recommend GNS as a possible fallback from OnioNS.

4.2.3 Namecoin

Namecoin is an early fork of Bitcoin [24] and is noteworthy for achieving all three properties of Zooko's Triangle. Namecoin holds digitally-signed information transactions in a data structure known as a block; each block links to a previous block, forming a public ledger known as a blockchain. Storing textual information such as a domain registration consumes some Namecoins, a unit of currency. 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.

While Namecoin is often advertised as capable of assigning names to Tor hidden services, it has several practical issues that make it generally infeasible to be used for that purpose. First, to authenticate registrations, clients must be able to prove the relationship between a Namecoin owner's secp256k1 ECDSA key and the target hidden service's RSA key, and constructing this relationship is non-trivial. Second, Namecoin is typically a heavyweight DNS: it generally requires users to pre-fetch and then verify the blockchain, which is 2.45 GB as of April 2015 [25]. Third, although Namecoin supports anonymous ownership of information, it is non-trivial to anonymously purchase Namecoins, thus preventing domain registration from being truly anonymous. These issues prevent Namecoin from being a practical alternative DNS for Tor hidden service. However, our work shares some design principles with Namecoin.

CHAPTER 5

SOLUTION

5.1 Overview

We propose the Onion Name System (OnioNS) as an abstraction layer to hidden service addresses and introduce ".tor" as a new pseudo-TLD for this purpose. 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 between a meaningful second-level domain name and his .onion address. Without loss of generality, let this be "example.tor → onions55e7yam27n.onion". We introduce a proof-of-work scheme that requires Bob to expend computational and memory resources to claim "example.tor", a more privacy-enhanced alternative to financial compensation to a central authority. 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 DoS flood attack.

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.

Second, Bob uses a Tor circuit to anonymously transmit his Record to an authoritative short-lived random subset of OnioNS servers, known as the *Quorum*, inside the Tor network. The Quorum archive Bob's Record in a sequential public ledger known as a *Pagechain*, of which each OnioNS node holds their own local copy. Bob's Record is received by all Quorum nodes and share signatures of their knowledge with each other, so they maintain a common database.

Third, Alice, uses a Tor client to anonymously connect to a name server outside the Quorum but mirroring their database, then ask for "example.tor". Alice receives Bob's Record, verifies it signature and proof-of-work, and follows the association to "onions55e7yam27n.onion". As Bob's Record is self-signed using Bob's private key, Alice can verify the Record's authenticity. Finally, Alice uses this address and the Tor hidden service protocol to contact 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 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 a .tor pseudo top-level domain (TLD) that is not used on the Internet DNS. The TLD is a name at depth one and is preceded by names at sequentially increasing depth. The term "domain name" refers to the identification 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.

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

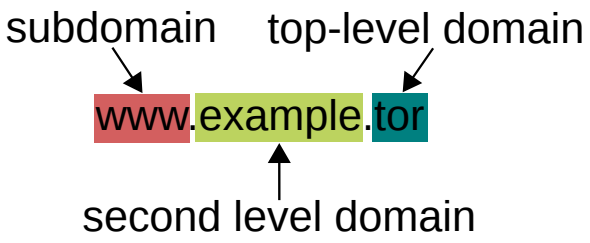

Figure 5.1: A sample domain name: a sequence of labels separated by delimiters. In OnioNS, hidden service operators build associations between second-level domain names and their hidden service address.

service operators claim a second-level name and then control all names of greater depth under that second-level name.

A **Record** is a small textual data structure that contains a single second-level domain name, a mapping of subdomains to .tor or .onion pseudo-TLD destinations, proof-of-work, a digital signature, a public key, and an optional PGP fingerprint. 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 **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.

A **Mirror** is any machine (inside or outside of the Tor network) that has performed a synchronization (section 5.6.2) 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 main page-chain.

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.

The **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.

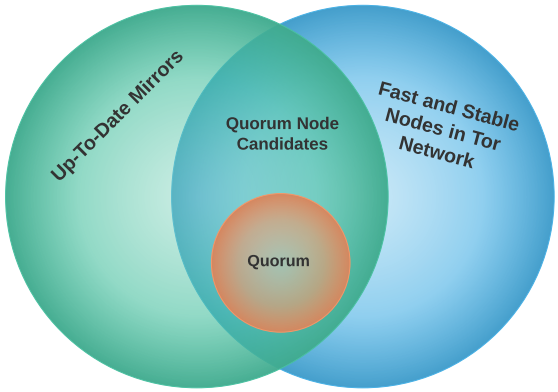

Figure 5.2: The relationship between Mirrors, Quorum Candidates, and the Quorum. A Mirror is any machine that holds the OnioNS Pagechain, Quorum Candidates are both up-to-date Mirrors and reliable Tor nodes, and the Quorum is randomly selected from the pool of Quorum Candidates.

Throughout the rest of this document, let Alice be a Tor client, and Bob be the operator of a hidden service with access to his private HS RSA key. As both Alice and Bob are Tor users, they can obtain and fully verify any past or current set of the three consensus documents described in section 1.2.2.

5.3 Basic Design

5.3.1 Claim on a Domain Name

To claim a domain name for his hidden service, Bob first generates a valid Record and transmits it over a Tor circuit to a randomly-selected Quorum node. Bob receives a response indicating whether the Record was accepted or not and can confirm that it was

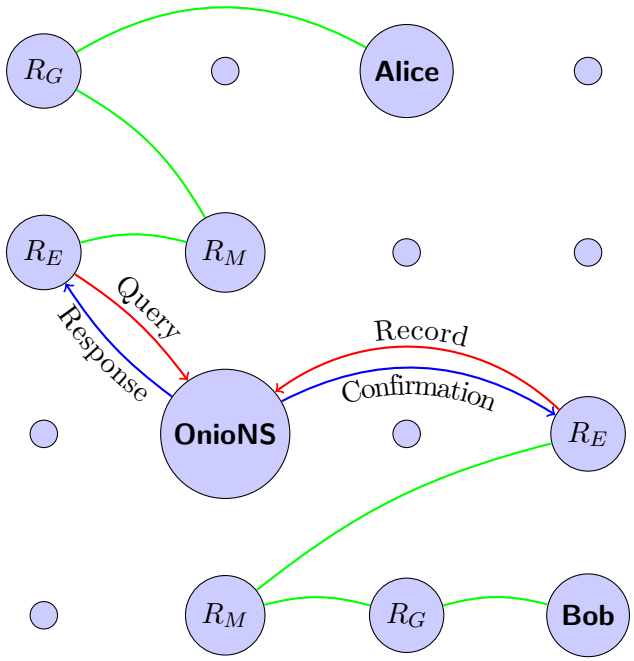

Figure 5.3: Bob uses a Tor circuit to anonymously upload a Record to OnioNS. Alice uses her own Tor circuit to query the system for a domain name, and she is given Bob's Record in response. Then Alice connects to Bob by Tor's hidden service protocol.

re-transmitted by immediately querying another Quorum node for that Record. Mirrors can subscribe to network events from other Mirrors in a peer-to-peer fashion, thus allowing Records and other information to quickly propagate the network, as illustrated in Figure 5.4.

5.3.2 Pagechain Maintenance

We introduce the Pagechain, a fundamental data structure in OnioNS, in order to keep all participating nodes in synchronization and to save Records in long-term storage. It is a distributed append-only transactional database of a fixed maximum length and is held locally by all Mirrors. Like a blockchain, the Pagechain is designed as a public and fully confirmable data structure – anyone can confirm the integrity, uniqueness, and validity of all data structures contained within. The head of the chain (the latest Page) is maintained by members of the current Quorum. Assuming that the entire Quorum is honest and maintains perfect communication, all Quorum nodes would be maintaining an identical

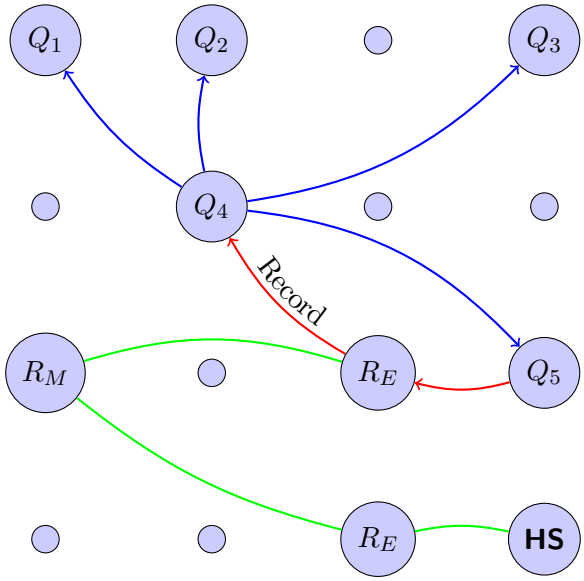

Figure 5.4: Bob uses his existing circuit (green) to inform Quorum node Q_4 of the new Record. Q_4 then sends his Record to all other Quorum nodes. Each node stores it in their own Page for long-term storage. Bob confirms from another Quorum node Q_5 that his Record has been received.

Page. However, due to malicious modifications or missed communication, disagreements may inevitably form within the Quorum. To avoid these disagreements from misleading the network, we let the network follow the Page maintained by the largest number of Quorum nodes, thus allowing the structure to be partially self-healing as illustrated in Figure 5.5.

5.3.3 Client Request

Typically, Alice will look up a domain name, performing a *domain query*. These are relatively straightforward. Let us assume that Bob, who owns "onions55e7yam27n.onion," has uploaded a Record containing "example.tor" and that another party Dave has a Record which claims "example2.tor" and that contains "sub.example2.tor" → "example.tor." Then Alice can query for "sub.example2.tor."

Alice's client-side software recognizes the .tor pseudo-TLD and distinguishes requests containing it from normal Internet lookups. Her software directs the request through a Tor circuit to some Mirror, O_r. For security and load reasons, Quorum nodes refuse to respond to queries, so Alice cannot ask them. O_r finds "sub.example2.tor" and returns Dave's

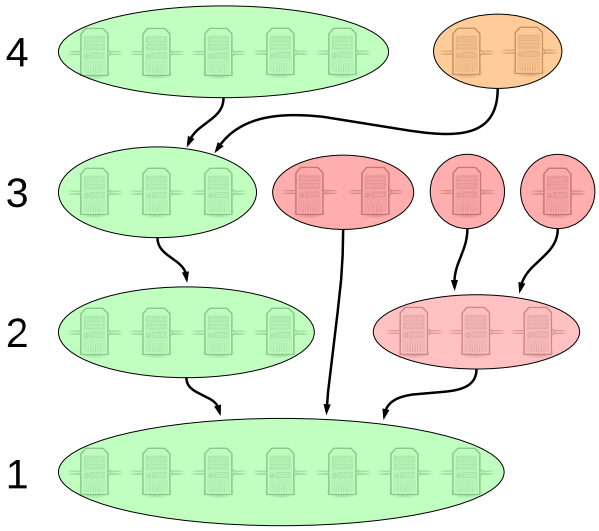

Figure 5.5: An example Pagechain across four Quorums with three side-chains. Quorum_1 is honest and maintains reliable flooding communication, and thus has identical Pages. Here, red nodes are colluding maliciously, orange missed some communication, and green nodes are acting honestly. Despite the disagreements, across all four days the largest clusters are honest nodes and thus integrity remains in the master Pagechain.

Record. Alice sees the "sub.example2.tor → example.tor" association and now queries O_r for "example.tor" since that domain is not in Dave's Record. O_r returns Bob's Record and Alice sees that "example.tor" is claimed by Bob and can connect to "onions55e7yam27n.onion" via the Tor hidden service protocol. She can send her original request of "sub.example2.tor" to Bob, allowing Bob's web server to provide specific content based on that hostname.

In this way, Alice can query a name server and load a hidden service by a meaningful name. We suggest optimizations and security enhancements to this protocol in Section 5.7.

5.4 Primitives

5.4.1 Cryptographic

OnionNS makes use of cryptographic hash algorithms, digital signatures, proof-of-work, and a pseudorandom number generator. We require that Tor routers generate an Ed25519 [26] keypair and distribute the public key via the consensus document. We note that while we can theoretically use existing NTor keys for digital signatures as it is possible to convert

Curve25519 to Ed25519 in constant time, we refrain from this because it is likely not a cryptographically secure operation. Therefore we require Tor to introduce Ed25519 keys to all Tor routers. If this is infeasible, Ed25519 can be substituted with RSA in all instances.

- Let $H(x)$ be a cryptographic hash function. In our reference implementation we define $H(x)$ as SHA-384.

- Let $S_{RSA}(m, r)$ 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 $S_{RSA}(m, r)$ use $H(x)$ as a digest function on m in all use cases. In our reference implementation we define $S_{RSA}(m, r)$ as EMSA PKCS1 1.5.

- Let $V_{RSA}(m, E)$ validate an RSA digital signature by accepting a message m and a public key R, and return true if and only if the signature is valid.

- Let $S_{ed}(m, e)$ be an Ed25519 digital signature function that accepts a message m and a private key e and returns a 64-byte digital signature. Let $S_{ed}(m, e)$ use $H(x)$ as a digest function on m in all use cases.

- Let $V_{ed}(m, E)$ validate an Ed25519 digital signature by accepting a message m and a public key E, and return true if and only if the signature is valid.

- Let PoW(i) be a one-way collision-free function that accepts an input key k and returns a deterministic output. 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 [27] [28]. 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, which is widely used throughout most programming languages and is well known for its speed, long period, and the high quality of its pseudorandom output [29].

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.

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. In each case, *type* is set to the Record type.

Create

A Create Record consists of nine components. 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".
name	Yes	The second-level domain name for this hidden service.
nameList	Yes	An array list of zero or more .tor subdomains and their destinations. Destinations use either .tor or .onion TLDs.
contact	No	Bob's PGP key fingerprint if he has one. Client can use this to contact Bob over encrypted email.
consensusHash	Yes	The hash of the consensus document that generated $Quorum_q$
nonce	Yes	Four random bytes.
pow	Yes	16 bytes that store the output of PoW(i).
recordSig	Yes	The output of $S_{RSA}(m, r)$ where $m = nameList \parallel timestamp \parallel consensusHash \parallel nonce \parallel pow$ and r is the hidden service's private RSA key.
pubHSKey	Yes	Bob's public RSA key.

Modify

A Modify Record allows an owner to update his registration with updated information. The owner corrects the fields, updates *consensusHash*, revalidates the proof-of-work, and transmits the record. Modify Records have a difficulty of $\frac{\text{difficulty}_{\text{Create}}}{4}$. Modify Records also act as Renew Records.

Move

A Move Record is used to transfer ownership of the second-level domain name and all associated subdomains from one hidden service key to another. Move Records must not contain modifications and must contain one additional field: *destPubKey*, the public key of the new owner. In this way, transfers are similar to Namecoin. Move records also have a difficulty of $\frac{\text{difficulty}_{\text{Create}}}{4}$.

Renew

Second-level domain names (and all associated subdomains) expire every $L_P \Delta q$ days because the Pagechain has a maximum length of L_P Pages. Renew Records must be reissued periodically at least every $L_P \Delta q$ days to ensure continued ownership of the domains contained within them. No modifications to existing domain names can be made in Renew records, and the domain names contained within must already exist in the Pagechain. Similar to the Modify and Move records, Renew records have a difficulty of $\frac{\text{difficulty}_{\text{Create}}}{4}$.

Delete

A Delete Record is used to relinquish ownership rights over a second-level domain name. This is useful if the operator feels that his private key has been compromised or if he has no further user for his domain. This issuance of this Record immediately triggers a purging of the domain name in the system, making it almost immediately available for others. There is no difficulty associated with Delete records, so they can be issued instantly.

5.5.2 Page

Each page contains five fields.

prevHash $H(prevHash \parallel recordList \parallel consensusDocHash)$ of a previous page.

recordList An array list of Records, sorted in a deterministic manner.

consensusDocHash $H(cd)$.

fingerprint The Tor fingerprint of the relay maintaining this Page.

pageSig The output of $S_{ed}(H(prevHash \parallel recordList \parallel consensusDocHash)], e)$ where e is the router's private Ed25519 key.

5.6 Protocols

In section 1.2.2 we described the three consensus documents that Tor routers and clients have: *cached-certs*, *cached-microdesc-consensus*, and *cached-microdescs*. Throughout this section, let "consensus documents at time X and day Y" specifically refer to these three documents when *valid-after* is set to X and Y. For protocols that specify a hashing of the consensus documents, let the hash only cover *cached-certs* and *cached-microdesc-consensus*; although a router's descriptor is split between *cached-microdesc-consensus* and *cached-microdescs*, the microdescriptors in *cached-microdesc-consensus* include the SHA-256 hash of the entire descriptor. All parties can obtain these consensus documents from any sources because *cached-certs* contains the signing keys that validate *cached-microdesc-consensus*. In practice, it is often efficient to compress these documents en-masse before transmission: they achieve very high compression ratios under Lempel-Ziv-Markov chain algorithm (LZMA).

5.6.1 Hidden Services

Record Generation

As invalid Records will be rejected by the network, Bob must generate a valid Record before broadcast:

1. Bob selects the value for *type* based on the desired operation.
2. Bob selects a second-level domain name for his hidden service and assigns it to *name*.
3. Bob defines subdomains and their destinations, constructing *nameList*.
4. 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.
5. Bob sets *consensusHash* to the output of $H(x)$, where x is the consensus documents 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*. Bob sets *pow* as PoW(*central*).
8. Bob sets *recordSig* as the output of $S_{RSA}(m, r)$ where $m =$ *central* $\|$ *pow* and r is Bob's private RSA key.
9. 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*$) \leq 2^{difficulty}$ where *difficulty* is a fixed constant that specifies the work difficulty. An example of a completed and valid record is shown in Figure 5.6.

Record Validation

Let Carol be a Tor client or a Mirror that receives a Record from another Mirror.

1. Carol checks that the Record contains valid JSON and that *type* is either "Create", "Modify", "Move", "Renew", or "Delete".
2. Carol checks that *nameList* is has a length $\in [1, 24]$ and that all subdomains have a second-level domain name of the *name* field. Additionally, Carol checks that there is

```
0  {
1      "type": "Create",
2      "name": "example.tor",
3      "subd": {"sub": "onions55e7yam27n.onion"},
4      "contact": "AD97364FC20BEC80",
5      "consensusHash": "uU0nuZNNPgilLlLX2n2r+sSE7+N6U4DukIj3rOLvzek=",
6      "nonce": "AAAABw==",
7      "pow":"iOuFHz+eBoxsxDDuX/torg==",
8      "recordSig": "bQCtRgJKvkG1Me1Nb0jB1cjN945vHyunFYesi7ildOrYeeZAZLWhf9
           azi7YhgL+V/9edPxlGX8+8AMTmrJp6DMWeBVZegANNDzTxkc//x72w88uenQcff
           JgEKZ1CyBFT3QxtIJvtsd/Te8Hwd60mnAxDR/42rD1QwhJ6PPoOCtc=",
9      "pubHSKey": "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDaXlifPm7dQkw0b
           F7tOeEdMT9QGM2xoKRZGkNtI8+qeaqx6eiqynVPuS4DYTbr3NppqG7cykteOJlY
           jBqcDPeaNIytos8q6qYyNEd3VuV6Mm46SL66BL/fIKMxmPNXLp8LfnY
           UzcUxtdMetxv64Q1Nh46NX6Z8579AXVue3TuKKwIDAQAB"
10 }
```

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.

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.

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

4. Carol checks the validity of *recordSig* against *pubHSKey* and *central* $\|$ *pow*.

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

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

7. 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.
2. Bob constructs a circuit, c_1, to a Mirror node m_1.
3. Bob asks for and receives from c_1 the $h_j = H(prevHash \| recordList \| consensusDocHash)$ hash and *pageSig* (the Ed25519 signature on that hash) from each Quorum node.
4. Bob confirms that $V_{ed}(h_j, E)$ returns true for each h_j 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 uses c_1 to ask m_1 for his second-level domain name.
8. Bob has confirmation that his Record was accepted and processed by the Quorum if m_1 returns the Record he uploaded.

5.6.2 OnioNS Servers

Let Charlie be the name of an OnioNS Mirror. Charlie listens for incoming Records or signatures from one or more other Mirrors.

Database Initialization

1. Charlie creates an initial Page P_{curr} and sets the $P_{prevHash}$ back-reference based on the Page Selection protocol.
2. Charlie sets *recordList* to an empty array.
3. Charlie downloads from some remote source 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.
4. Charlie sets $consensusDocHash = H(cd)$.

```
0 {
1       "prevHash": 0,
2       "recordList": [],
3       "consensusDocHash": "uU0nuZNNPgilLlLX2n2r+sSE7+N6U4DukIj3rOLvzek=",
4       "fingerprint": "2FC06226AE152FBAB7620BB107CDEF0E70876A7B",
5       "pageSig": "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/kBEpyXRSopzjo
            Fs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh+QOnKl0fKBN7
            fqowjkQ3ktFkR0Vuox9WrrbNTMa4+up0Np52hlbKA3zSRz4fbR9NVlh6uuQ="
6 }
```

Figure 5.7: A sample empty Page.

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

6. Charlie sets $pageSig = S_{ed}(H(prevHash \parallel recordList \parallel consensusDocHash), e)$.

Page Selection

The Page Selection protocol relies on our security assumption that the largest set of Quorum nodes with agreeing and valid Pages are acting honestly. In this protocol, Charlie chooses the Page P_c maintained by that set.

1. Charlie calculates the Quorum via the Quorum Derivation protocol.

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

3. For each Page,

   (a) Charlie checks that *prevHash* references some page from the previous Quorum.

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

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

   (d) Charlie checks that $V_{ed}(h, E)$ returns true.

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

5. For each set of Pages with equal h,
   (a) Charlie checks that $consensusDocHash = H(cd)$.
   (b) Charlie checks each Record in *recordList* via the Record Validation protocol.
6. If the validation of a Page fails, Charlie removes it from the equal-h list.
7. 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_i from the oldest available q_j to the most recent q_k,

1. Charlie chooses a Page P_{c2} from $Quorum_{q_i}$ via the Page Selection Protocol.
2. Charlie checks that $f(P_c, P_{c2}, q)$ returns true, otherwise repeats step 1 to choose another Page from the next largest set of Pages that have equal h.
3. Charlie calculates $h_i = H(prevHash \,\|\, recordList \,\|\, consensusDocHash)$, fields from P_{c2}.
4. Charlie checks that P_{c2}'s *prevPage* field equals h_{i-1} or that $i = j$, otherwise repeats step 1 to choose the Page from the next largest set.

Synchronization

1. Charlie randomly selects a Mirror relay, 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 Mirror R_k, $k \neq j$, and downloads the invalid Pages from R_k.

4. Charlie randomly selects a Quorum node Q_c and subscribes to Record and signature networking events from it.

5. 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 and are responsible for handling incoming Records from hidden service operators. 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 [1].

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 [1]: 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 number of minutes elapsed in that day. 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.

We suggest placing t inside a new field within the router descriptor in future work, but our use of the Contact field eases integration with existing Tor infrastructure. The field is 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 r is received, Q_j

1. Q_j rejects r if the Record is not valid.

2. Q_j rejects r if no such hidden service descriptor exists in Tor's distributed hashtable.

3. Q_j rejects r if Q_j's Pagechain contains ≥ 2 Create Records containing r's *pubHSkey*.

4. If r is a Create record, Q_j rejects r if its second-level domain already exists in Q_j's Pagechain.

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

   (a) r's Create Record was not found in the Pagechain.

   (b) r's *pubHSKey* does not match the latest Record found in the Pagechain under its second-level domain name.

6. If Q_j has rejected r, Q_j informs Bob of this outcome and its reason.

7. If Q_j has not rejected r, Q_j informs Bob that r was accepted and Q_j merges the Record into its 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, since Quorum nodes don't answer queries.

Quorum Derivation

1. Alice obtains the consensus documents, *cd*, published on day $\lfloor \frac{q}{\Delta q} \rfloor$ at 00:00 GMT.

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

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

4. Alice uses f to randomly scramble *qc*.

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

Domain Query

The basic design of the Domain Query is relatively straightforward.

1. Alice constructs a Tor circuit to Charlie.

2. Alice provides a .tor domain name d into the Tor Browser, which is treated as a special case by her client software.

3. If d's highest-level name is "www", Alice's software transparently removes that name.

4. Alice asks Charlie for the most recent Record r containing d.

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, Alice throws an assertion error.

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 r's key in Tor's distributed hash table.

10. Alice sends the original d to the hidden service.

We supplement this protocol with an additional data structure in section 5.7.3 as a defence against Record forgeries. It is also possible for Alice to request and the Page p containing r and all digital signatures from each Quorum node, allowing Alice to perform width verification as she can see that a large percentage of Quorum nodes maintained p.

Of course, Alice can be certain that the Record r she receives is authentic and that d 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 MiBs [1], 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 whose *pubHSKey* hash to Alice's address. We note that 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 a Merkle Tree to prevent Mirrors from forging Records or falsely claiming non-existence, thus preventing them from being actively malicious in all significant cases.

5.7.1 AVL Tree

An AVL tree is a self-balancing binary search tree with $\mathcal{O}(\log(n))$ time for search, insertion, and deletion operations. We suggest that OnioNS mirrors 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 builds an AVL tree. The leaves of the tree are the location of each Record in the Pagechain, while the keys are the Record's *name* field. The Mirror should then update the AVL tree when it receives new Records that were recently sent to Quorum nodes. Create Records trigger an insert operation, Delete Records cause a deletion, and all other Records update a location pointer to the more recent Record. This approach effectively transforms the lookup time of Records for Domain Queries from $\mathcal{O}(n)$ to $\mathcal{O}(\log(n))$ in the average and worst cases.

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 32 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 Merkle Tree

A Merkle tree is a hash tree, a special type of binary tree wherein each non-leaf node is the hash of node's children. We introduce the Merkle tree for two purposes: 1) to prove the non-existence of a domain name, and 2) to prove the authenticity of an existing domain name. 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 (the Quorum) can sign the Merkle tree once.

Merkle trees also allow anyone to verify that a leaf node is part of a given hash tree in $\mathcal{O}(\log(n))$ and without requiring knowledge of the entire tree. We utilize these two properties to allow clients to authenticate Records with minimal networking costs in a single query to an untrusted Mirror name server. To our knowledge this represents the first alternative DNS to authenticate denial-of-existence claims on domains en-masse. To achieve this, let each Quorum node

1. Construct an array list *arr*.
2. For each Record r in the Pagechain, add $r_{name} \parallel H(r)$ to *arr*.
3. Sort *arr*.
4. Construct a Merkle tree T from *arr*.

5. Generate $sig_T = S_{ed}(t \parallel r, e)$ where t is a timestamp and r is the root hash of T.

As Records contains all subdomains under a single second-level domains, T needs only contain r_{name} to reference all domains in r, which further saves space. Then during a Domain Query Alice may use T to authenticate a domain d and verify non-existence for a Record r.

1. Alice extracts the second-level name c from d.

2. If r exists, Charlie returns the leaf node containing r and all the tree nodes from leaf r to the root and their sibling nodes, so that Alice can verify authenticity of r by recomputing the root hash and verify that the largest subset of Quorum nodes signed the same root hash.

3. If Charlie claims non-existence of c, he returns two adjacent leaves a and b (and the nodes on their paths and siblings) such that $a < c < b$, or in the boundary cases that a is undefined and b is the left-most leaf or b is undefined and a is the right-most leaf.

4. If either assertion fails, Charlie is dishonest.

The Quorum must regenerate T every ΔT hours to include new Records. Then Alice needs only fetch the signatures on T at least every ΔT hours to ensure that she can authenticate new Records during the Domain Query. Although new Records can traverse the network nearly instantaneously, Alice cannot authenticate or verify denial-of-existence claims on Records newer than ΔT. Alice must also fetch the L_Q signature from all Quorum nodes and assert that T is signed by the largest set of nodes maintaining the same Page so as to agree with our last security assumption.

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. 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. It is also impossible to determine which Tor routers are under Eve's control and which are honest in advance, so we 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 lower 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 assumption on the security of Tor circuits.

We assume 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 $L_E > \frac{L_Q}{2}$. The second scenario can be statistically 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.

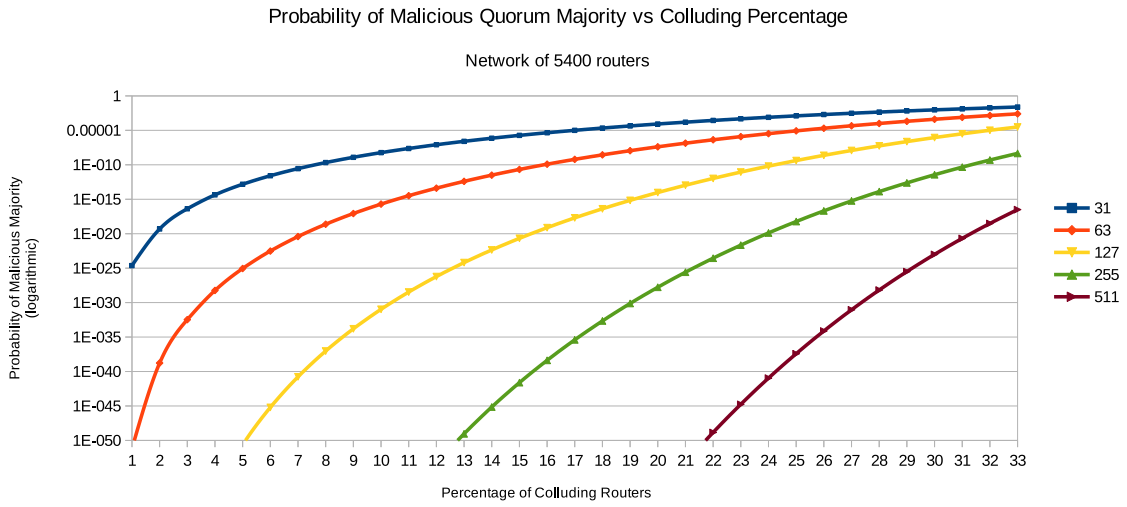

Figure 6.1: The probability that Eve controls the majority of the Quorum is given by the PMF of the hypergeometric distribution. We fix N at 5,400 nodes and graph Eve's success probability as a function of an increasing percentage of Eve-controlled colluding routers. We examine five selections for L_Q: 31, 63, 127, 255, and 511. We do not consider percentages beyond 33 percent as those represent a near-complete compromise of the Tor network.

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 with approximately two percent collusion, although 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.

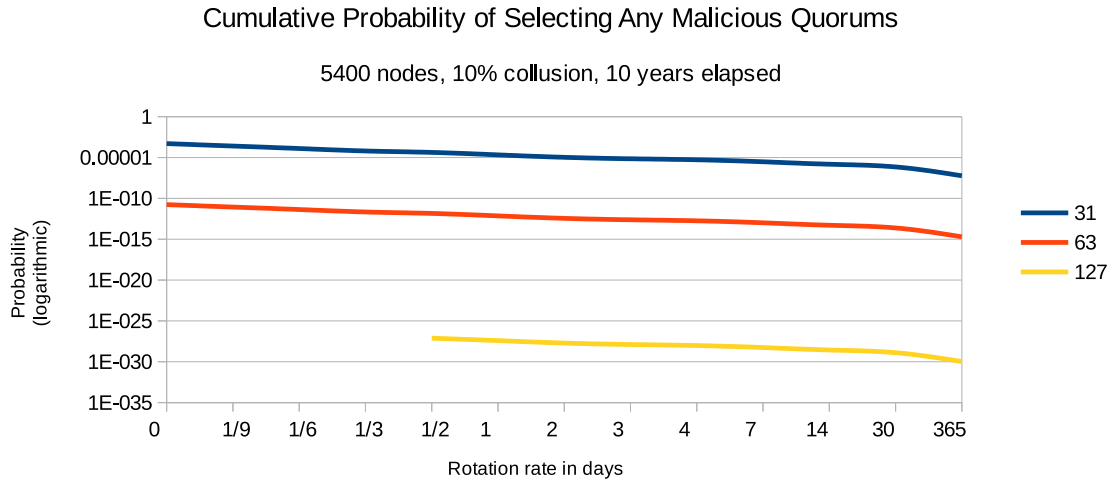

Figure 6.2: The cumulative probability that Eve controls any Quorum at different rotation rates. We assume 10 percent collusion in a network of 5400 Tor routers, and view across 10 years. We do not graph L_Q values of 255 or 511 as they generate probabilities far below our $10^{-38.532}$ threshold; $L_Q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

Figure 6.2 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, 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, 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.

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. However, the networking and performance load scales linearly with Quorum size. Based on this balance and our above analysis, we suggest 127 or 255 for values of L_Q and 7 or 14 for Δq.

6.1.2 Entropy of Tor Consensus Documents

We use Tor's consensus documents as a sources of entropy agreed upon by all parties, however we have not yet demonstrated that the network status contains enough entropy to provide reasonable assurance that Eve cannot guess the next Quorum in advance. If Eve could predict future Quorums, Eve can subvert the Quorum Derivation protocol in a variety of attack vectors. However, this would fail our security assumption against adaptive compromise in the presence of OnioNS. Rather than introducing defences against these attack vectors, we nevertheless believe that ensuring sufficient entropy in the consensus documents is a superior defence.

Periodically, Tor routers upload signed *descriptors* — routing information, cryptographic keys, and other information — to Tor's directory authorities (dirauths). Once per hour, the dirauths aggregate and republish the descriptors back to the network, enabling Tor's network to be dynamic and distributing new information to all parties in an efficient and timely manner. While Tor provides no guarantee that the network is using the same consensus at the same time, the consensus is timestamped, so we can reference it by their time of publication. We focus on two essential documents that clients assemble from the consensus: *cached-certs*, a list of long-term identity and short-term signing RSA keys from each dirauth; and *cached-microdesc-consensus*, essential information about each router, such as networking information and capabilities. Since the dirauths publish a new consensus every hour, we must analyse the entropy rate between consecutive hours.

We constructed a first-order Markov model and estimated the entropy of the *cached-microdesc-consensus* document by analysing the transitions for various fields between consecutive consensuses. We also provide analysis on the dynamics of the Tor network by counting routers that enter or leave the Tor network between consecutive consensuses in Figure 6.3. Here we identity routers by their identity keys; just as Tor does, we consider routers that change their identity keys as two different routers; one leaving and one entering the network. Routers entering the network introduce significant amounts of information into the consensus. For example, *cached-microdesc-consensus* contains the 160-bit SHA-1 hash of the router's RSA-1024 identity key, which are generated through OpenSSL. Thus new routers contribute at least 160 bits of information. Figure 6.3 suggests that we may expect approximately 100 to 175 routers to leave or enter the network per hour. Thus we may expect approximately 16,000 to 28,000 bits of entropy per hour from this dynamic.

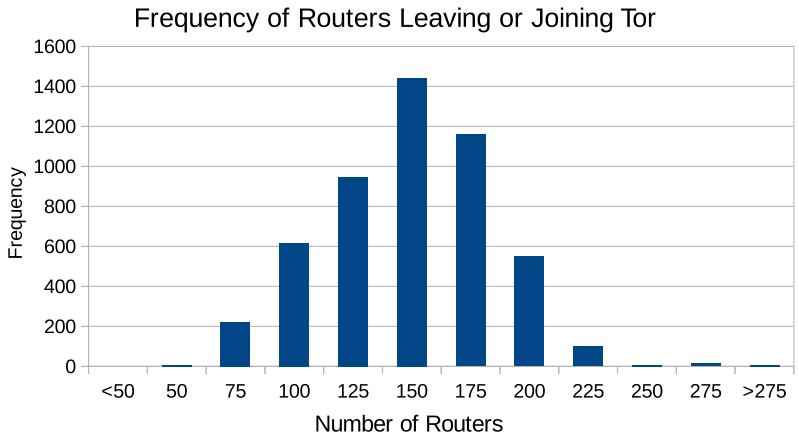

Figure 6.3: A histogram of the number of routers entering or leaving the network between consecutive consensuses across the seven-month period.

For routers that are present between consecutive pairs, we focus on six critical fields from each router's descriptor inside *cached-microdesc-consensus*: *nickname*, the router's name chosen by its operator or a default name; *publication*, the time when it last published a descriptor; *IP address*, its network address; *ORPort*, the network port for onion routing; *version*, the version of the Tor protocol that this relay is running; and *bandwidth*, as self-

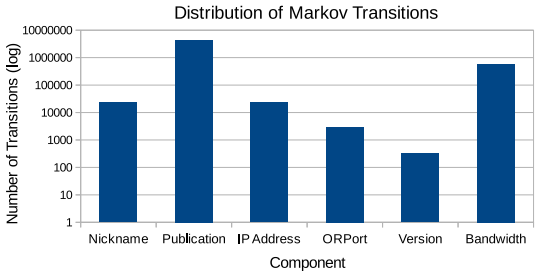

Figure 6.4: The number of observed transitions for *nickname*, *publication*, *IP address*, *ORPort*, *version*, and *bandwidth* between consecutive consensuses across the seven-month period.

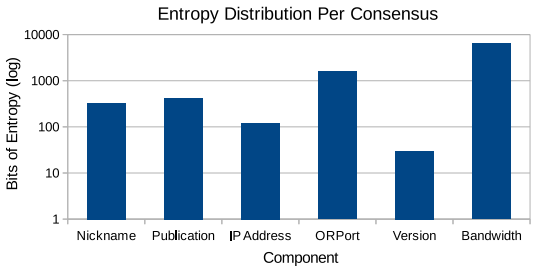

Figure 6.5: The entropy rate distribution for each of the six fields in *cached-microdesc-consensus*, scaled by the average size of the Tor network.

reported or as measured by the dirauths. Tor routers change *publication* when either 18 hours has elapsed since the last descriptor publication, its fields have changed, or if its uptime has been reset.

We obtained from `collector.torproject.org` 5,067 archived hourly publications of *cached-microdesc-consensus* across a seven-month period between September 1, 2014 00:00 UTC and March 31, 2015 23:00 UTC. We construct a Markov model for the six aforementioned fields and illustrate the number of observed transitions for each field in Figure 6.4.

Then the entropy rate is given by equation 6.1, where P_i is the probability of state i and $P_i(j)$ is the probability of state j given i (the i-j transition). We multiply the entropy rate by the total number of routers in the Tor network. This assumes that router's are independently and uniformly distributed, but this is not always the case as small sets of routers may be managed together and change as one. However, identifying these sets and analysing them separately is non-trivial to impossible; administrators may operate anonymously or try to purposely hide their management of multiple geographically-distance routers. Due to this, this assumption results in an estimation of the entropy and not an exact value. We calculate this rate estimation for each of the six fields and illustrate the results in Figure 6.5.

$$\mathrm{H}(\mathcal{S}) = -L_T \sum_i P_i \sum_j P_i(j) \log_2 P_i(j) \tag{6.1}$$

The average size of the Tor network across the seven-month period was 6,672 routers, thus together these fields contributed on average approximately 8,896.7 bits of entropy per hour across our seven-month view. The two documents contain at least this much entropy; our analysis does not comprehensively cover all the fields in *cached-certs* and *cached-microdesc-consensus*, so other fields may also contribute additional information. Based on this analysis, the approximately 16,000 to 28,000 entropy bits contributed hourly by routers joining or entering the Tor network, and our previous statistical calculations, we conclude that with $L_Q \geq 127$, the Quorum Formation protocol is secure under our design assumptions. Our analysis shows that the size and dynamic nature of the Tor network provides the strongest defence against Quorum-level attacks.

Malicious Entropy Reduction

The Quorum Derivation protocol describes initializing the Mersenne Twister with a 384-bit seed. If we assume that Eve desires that the Quorum Derivation protocol produce a Quorum pleasing to Eve (such as including her malicious routers in the Quorum or rejecting specific honest routers from the Quorum) based on some hash k. As $H(x)$ is SHA-384, SHA-384's strong resistance to preimage attack forces Eve to spend $L_T * 2^{383}$ operations on average to find k. Eve may also try to manipulate her router's descriptors such that $H(cd_{q+1}) = k$, but SHA-384's resistance to second-preimage attacks also requires this to require $L_T * 2^{383}$ operations as well. The number of operations involved in this attack vector is significantly more than the operations involved in breaking AES, so we disregard the possibility of manipulating the Quorum Derivation protocol in this way.

6.1.3 Sybil Attacks

Eve may also attempt to increase her probability of including her malicious nodes in the Quorum via Sybil attacks. We offer no defence against this type of attack, although Tor does. The attack is difficult to carry out in practice due to the slow build of trust within the Tor network. Directory authorities would give Eve's nodes the Fast and Stable flags after weeks of continual uptime and a history of reliability. For large-scale Sybil attacks,

this introduces a significant time and financial cost to Eve. We also note that choices of L_Q and Δq also offer significant statistical defences against Sybil attacks, as illustrated in Figures 6.1 and 6.2, shown above.

6.1.4 Hidden Service Spoofing

OnioNS does not require a hidden service operator to reveal any personally-identifiable information. Hidden services are only known by their public key and domain names, and we assume that the hidden service is authentic. We have also observed spoofed hidden services in the wild, suggesting that this problem already exists in Tor's environment. We do not introduce a reputation system or distributed verification system, and note that it is difficult if not impossible to construct a reliable defence against hidden service spoofing attacks due to their anonymous nature.

One possible solution, although it severely compromises anonymity, is to register a hidden service with an SSL Certificate Authority and apply an SSL certificate to the server. In this way, TLS communication provides an authenticity check against the hidden service, although TLS also sets up a redundant encryption layer that may decrease performance. However, in practice this solution is very rarely seen; out of approximately 25,000 hidden services [1] [2], to date only three hidden services have 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. In future work we may study the security implications of signing the .onion address or the .tor OnioNS domain name. Due to their severe privacy concerns and the security controversy surrounding the centralized CA Chain of Trust, generally speaking we do not recommend the application of SSL certificates to Tor hidden services.

6.1.5 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 Record Generation protocol with the objective of requiring the hidden service operator to also perform the scrypt proof-of-work. However,

our protocol does not entirely prevent the operator from outsourcing the computation to secondary resource in all cases.

Let Bob be the hidden service operator, and let Craig be a secondary 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*, *name*, *nameList*, *contact*, and *consensusHash* fields.
2. Bob sends R to Craig. Let *central* be $type \| name \| nameList \| contact \| consensusHash \| nonce$.
3. 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.
4. Craig sends all $C_{0 \leq j \leq K}$ to Bob.
5. 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 $S_{RSA}(m, r)$ where $m = central \| pow$ and r is Bob's private RSA key.
   (c) Bob has found a valid record if $H(central \| pow \| recordSig) \leq 2^{difficulty}$

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.

6.1.6 DNS Leakage

Human mistakes can also compromise user privacy. One such security threat is the accidental leakage of .tor lookups over the Internet DNS. This vulnerability is not limited to OnioNS and applies to any alternative DNS; users may mistakenly attempt lookups over the traditional Internet DNS. Mohaisen and Thomas observed .onion lookups on root DNS servers at a frequency that corresponded to external global events, highlighting the human factor in those leakages [30]. For OnioNS, this may occur if their client software was not properly configured, if their browser was not properly configured to or could not communicate with Tor, or for other reasons. We offer no defence against this attack vector and note that any defence against it would need to introduce lookup whitelists or blacklists into common browsers such as Chrome, Firefox, and Internet Explorer to prevent them from attempting lookups for pseudo-TLDs.

6.2 Objectives Assessment

OnioNS achieves all of our original requirements. OnioNS Records do not contain any personal or location information and the PGP key is optional and can be anonymous. OnioNS performs Domain and Onion Queries through Tor circuits, so resolvers cannot sufficiently distinguish users to track their lookups. The system uses self-signed Records and a Merkle tree to provide end-to-end authentication, which the client can obtain in a single query with lightweight bandwidth and CPU costs. Everyone holding the Pagechain can verify that the Records are unique and reject the introduction of collisions. The system is distributed throughout the Tor network and the selection of authoritative Quorum nodes is random and temporary, decreasing the load, responsibility, and attack potential for any single node. OnioNS is an optional add-on to Tor, so the system is backwards compatible. Finally, as we demonstrate in the next chapter, OnioNS is also easy-to-use as it loads a hidden service transparently and without requiring any input from the user.

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 the network.

CHAPTER 7

IMPLEMENTATION

7.1 Reference Implementation

Alongside this publication, we provide a reference implementation of OnioNS. We utilize C++11, the Botan cryptographic library, the Standard Template Library's (STL) implementation of Mersenne Twister, and the libjsoncpp-dev library for JSON encoding. We develop in Linux Mint and compile for Ubuntu Vivid, Utopic, Trusty, and their derivatives. Our software is built on Canonical's Launchpad online build system and is available online on Github.

7.2 Prototype Design

We have developed an OnioNS prototype that implements the Record Generation and Domain Query protocols. In this initial prototype, we use a static server and a hidden service that we deployed at onions55e7yam27n.onion. We constructed a Record containing an association between "example.tor" and "onions55e7yam27n.onion". Next, we transmit this Record to the server over a Tor circuit. Finally, we modified Tor such that the following procedures occur client-side as illustrated in Figure 7.1:

1. The user enters in "example.tor" into the Tor Browser.
2. The Tor Browser sends "example.tor" to Tor's SOCKS port for resolution.
3. The Tor client intercepts "*.tor", places the lookup in a wait state, and sends "example.tor" to the OnioNS client over a named pipe.
4. The OnioNS client connects to the static OnioNS server over a Tor circuit and performs a Domain Query.

5. The server responds with the Create Record containing “example.tor”.

6. The client writes “onions55e7yam27n.onion” to the Tor client over another named pipe.

7. The Tor client resumes the lookup and rewrites the original “example.tor” lookup with “onions55e7yam27n.onion”.

8. The Tor client contacts the OnioNS hidden service and passes the webpage to the Tor Browser.

9. The Tor Browser displays the website contents and preserves the “example.tor” domain name.

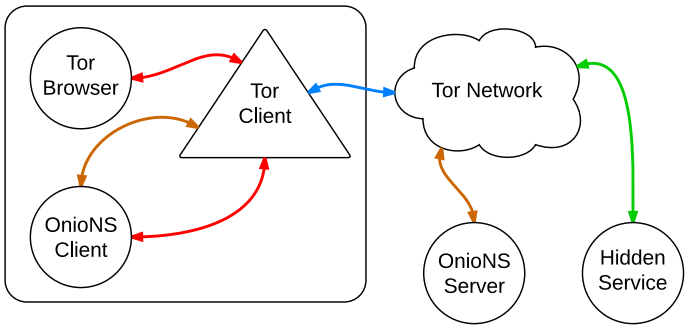

Figure 7.1: The overview of our OnioNS prototype. The Tor Browser passes an unknown .tor domain to the OnioNS through the Tor software (red) which resolves the domain anonymously over a Tor circuit (orange) to a remote resolver. Finally, the Tor software contacts the hidden service in the traditional way. (green) The Tor Browser communicates to the Tor client over its SOCKS port, while the OnioNS client communicates over named pipes (red) and Tor’s SOCKS port (orange).

Tor performs asynchronously here; it must place the lookup on hold, free its event loop to avoid a deadlock, and asynchronously resume the lookup once it receives the response. From the user’s perspective, the content of `http://onions55e7yam27n.onion` loads transparently under the `http://example.tor` URL.

7.2.1 Challenges

We encountered two significant challenges while implementing the prototype.

Our first modification to the Tor software used blocking I/O for communication with the OnioNS client. This caused Tor's event loop to pause while the OnioNS was resolving the domain name. When the OnioNS client attempted to use Tor to construct a circuit to the OnioNS server, Tor could not respond as it was waiting on I/O. This resulted in an unresolvable deadlock. After collaboration with several Tor developers we migrated our Tor modification to libevent, a software library that enables asynchronous event. Libevent is heavily used throughout Tor, Google Chrome, ntpd, and other software. Libevent enabled the OnioNS client to communicate over Tor, communicate with the remote OnioNS resolver, and return the hidden service address to Tor. Libevent then fired a callback method to contact the hidden service.

Our second challenge was telling Tor to place the resolution of the domain on hold. Previously, Tor would attempt to interpret the .tor domain and fail the lookup almost instantaneously. To resolve this, we placed the resolution in a waiting state. Then when OnioNS resolved the domain, our Libevent callback resumed the lookup, passed in the initial state, and allowed the lookup to continue as if a hidden service address had been requested in the first place. This allowed the Tor Browser to view the destination hidden service under a .tor domain name.

7.2.2 Performance

We conducted several performance measurements for the Record Generation and Domain Query protocols. Our experiment involves two machines, A and B. Machine A has an Intel Core2 Quad Q9000 (Penryn architecture) @ 2.00 GHz CPU from late 2008 and Machine B has an Intel i7-2600K (Sandy Bridge architecture) @ 4.3 GHz CPU from 2011. We chose machines with this hardware in order to represent low-end and medium-end consumer-grade computers, respectively. To minimize latency on our end, both machines were hosted on 1 Gbits connections at Utah State University. We hosted our hidden service and TCP server on Machine B. Then on Machine A, we measured the time required to connect to

our hidden service over OnioNS and over the more direct hidden service protocol.

We selected the parameters of scrypt such that it consumed 128 MB of RAM during operation. We consider this an affordable amount of RAM for low-end consumer-grade computers. We created a multi-threaded implementation of the Record Generation protocol and used all eight virtual CPU cores on Machine B to generate our Record. As expected, our RAM consumption scaled linearly with the number of scrypt instances executed in parallel; we observed approximately 1 GB of RAM consumption during Record Generation. We set the difficulty of the Create Record such that the Record Generation protocol took only a few minutes on average to conduct, but in the future we may change it so that the protocol takes several hours instead.

Processing Time

We measured the CPU wall time required for different parts of client-side protocols. We measured how long it takes the client to build a Record from a JSON-formatted textual string, which involves parsing and assembly of the various fields; the time to check the proof-of-work PoW, and the time to check the *recordSig* digital signature.

Description	**A Time (ms)**	**B Time (ms)**	**Samples**
Parsing JSON into a Record	0.052	0.0238	100
Scrypt check	896.369	589.926	25
Check of $S_d(m, r)$ RSA signature	0.06304	0.0267	200

Machine B, as expected, performed much faster than Machine A at all of these tasks. Parsing and signature checks both took trivial time, though the total time was dominated by the single iteration of scrypt. Record Validation protocol is single threaded and consumes 128 MB of RAM due to scrypt.

Latency

We compared the load latency between an OnioNS domain name with a traditional hidden service address. Our tests measured the time between when a user entered in

"example.tor" into the Tor Browser to the time when the browser first began to load our hidden service webpage. We also tested `http://onions55e7yam27n.onion`, the destination of "example.tor". We performed our experiment 15 times with a different client-side Tor circuit for each by restarting Tor at each iteration. To prevent browser-side caching, we restarted the Tor Browser between tests as well.

Lookup	**Fastest Time**	**Slowest Time**	**Mean Time**
.tor	6.1	8.5	7.1
.onion	9.3	12.2	10.2

The latency is circuit-dependent and heavily depends on the speed of each Tor router and the distance between them. To avoid the latency cost whenever possible, we implemented a DNS cache into the OnioNS client-side software to allow subsequent queries to be resolved locally, avoiding unnecessary remote lookups.

CHAPTER 8

FUTURE WORK

In future work we will expand our implementation of OnioNS and develop the remaining protocols. While our implementation functions with a fixed resolver, we will deploy our implementation onto larger and more realistic simulation environments such as Chutney and PlanetLab. When we have completed dynamic functionality and the remaining protocols, we will pursuit integrating our implementation into Tor. We expect this to be straightforward as OnioNS is designed as a plugin for Tor, introduces no changes to Tor's hidden service protocol, and requires very few changes to Tor's software. In future developments, Tor's developers may also make significant changes to Tor's hidden services, but OnioNS' design enables our system to become forwards-compatible after a few minor changes.

Additionally, several questions need to answered in future studies:

- Should the Quorum expire registrations that point to non-existent hidden services, and if so, how can this be done securely?
- How can we reduce vulnerability to phishing/spoofing attacks? Can OnioNS be adapted to include a privacy-enhanced reputation system?
- How can OnioNS support domain names with international encodings? A naïve approach to this is to simply support UTF-16, though care must also be taken to prevent phishing attacks by domain names that use Unicode characters that visually appear very similar.
- What other networks can OnioNS apply to? We require a fully-connected networked and an global source of entropy. We encourage the community to adapt our work to other systems that fit these requirements.

CHAPTER 9

CONCLUSION

We have introduced OnioNS, a Tor-powered distributed DNS that maps unique .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 vectors already well-understood throughout the Tor literature.

We use the Pagechain distributed data structure to prevent disagreements from forming within the network. Furthermore, every participant can verify the uniqueness of domain names. The 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] T. T. Project, "Tor metrics," https://metrics.torproject.org/, 2015, accessed 4-Feb-2015.

[2] G. Kadianakis and K. Loesing, "Extrapolating network totals from hidden-service statistics," *Tor Technical Report*, p. 10, 2015.

[3] B. Miller, L. Huang, A. D. Joseph, and J. D. Tygar, "I know why you went to the clinic: Risks and realization of https traffic analysis," in *Privacy Enhancing Technologies*. Springer, 2014, pp. 143–163.

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

[5] 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.

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

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

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

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

[10] L. Overlier and P. Syverson, "Locating hidden servers," in *IEEE Symposium on Security and Privacy*. IEEE, 2006, pp. 15–18.

[11] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker, "Low-resource routing attacks against tor," in *Proceedings, 2007 ACM Workshop on Privacy in Electronic Society*. ACM, 2007, pp. 11–20.

[12] 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.

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

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

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

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

[17] S. Nicolussi, "Human-readable names for tor hidden services," *Leopold-Franzens-Universitat Innsbruck, Institute for Computer Science*, 2011.

[18] 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.

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

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

[21] 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.

[22] M. Wachs, M. Schanzenbach, and C. Grothoff, "A censorship-resistant, privacy-enhancing and fully decentralized name system," in *Cryptology and Network Security*. Springer, 2014, pp. 127–142.

[23] D. J. Bernstein, "Dnscurve: Usable security for dns," http://dnscurve.org/, 2009, accessed 17-April-2015.

[24] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Consulted*, vol. 1, no. 2012, p. 28, 2008.

[25] bitinfocharts.com, "Crypto-currencies statistics," https://bitinfocharts.com/, 2015, accessed 17-April-2015.

[26] 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.

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

[28] C. Percival and S. Josefsson, "The scrypt password-based key derivation function," *Internet Engineering Task Force*, 2012.

[29] 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.

[30] M. Thomas and A. Mohaisen, "Measuring the leakage of onion at the root," *Proc. of the 12th Workshop on Privacy in the Electronic Society*, 2014.