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.

(88 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.

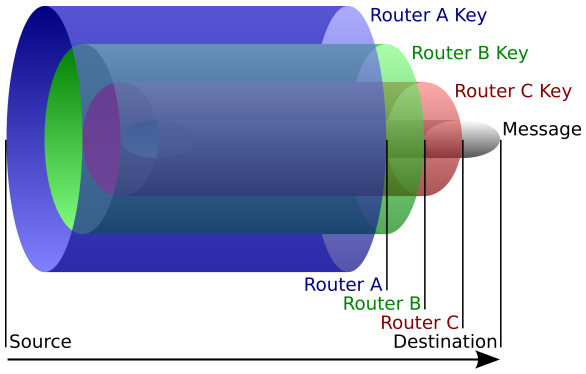

Figure 1.1: An example cell and message encryption in an onion routing scheme. Each router "peals" off its respective layer of encryption; the final router exposes the final destination.

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].

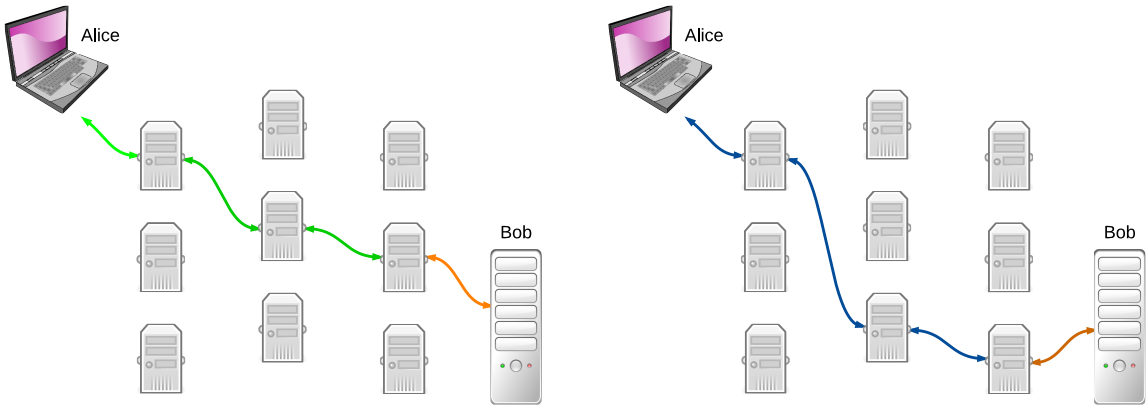

Figure 1.2: Alice communicates privately to Bob through a Tor circuit. Her communication path consists of three routers, an entry, middle, and exit. Although Bob's identity and location is known to Alice, the Tor circuit prevents Bob from knowing Alice's identity or location. At a later time, Alice may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

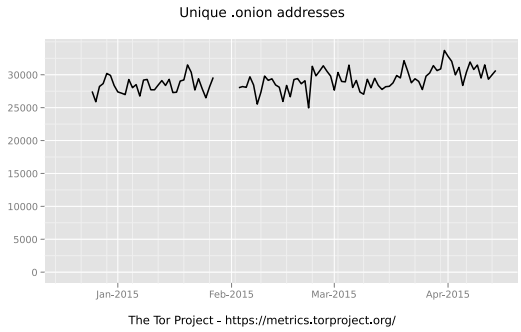

Figure 1.3: The number of unique .onion addresses seen in Tor's distributed hashtable between January through April 2015 [1] [2].

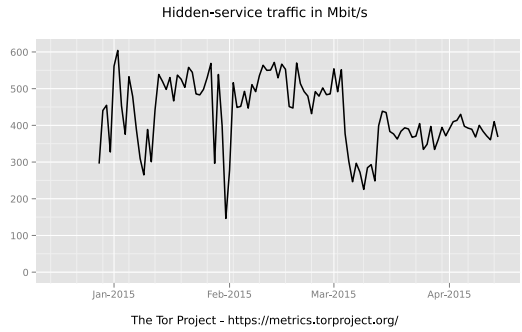

Figure 1.4: The amount of traffic generated by hidden services between January through April 2015 [1] [2].

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 privacy-enhanced queries.** Clients should be anonymous, indistinguishable, and unable to be tracked by name servers.

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].

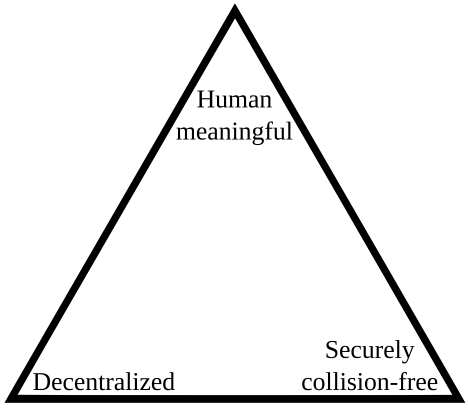

Figure 3.1: Zooko's Triangle.

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

In Section 2.2 we listed the design requirements that ideally must be met by a naming system in order to be applicable to Tor hidden services. We now examine several workarounds and existing naming systems against these objectives.

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 services; 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]. We consider this approach very similar to vanity key generation. 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 [25].

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 [26]. 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) a Tor-powered distributed naming system that maps .tor domain names to .onion addresses. The system has three main aspects: the generation of self-signed claims on domain names by hidden service operators, the processing of domain information within the OnioNS servers, and the receiving and authentication of domain names by a Tor client.

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

1. Significantly reduces the threat of record flooding.

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

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

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

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

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

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

5.2 Definitions

To discuss OnioNS precisely we must first define some central terms that we will use throughout the rest of this document. We detail their exact contents in section 5.5 and how they are used throughout sections 5.3 and 5.6.

domain name

A *domain name* is a case-insensitive identification string claimed by a hidden service operator. The syntax of OnioNS domain names mirrors the Internet DNS; we use a sequence of name-delimiter pairs with 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 identiciation string as a whole, while "second-level domain" refers to the central name that is immediately followed by the TLD, as illustrated in Figure 5.1. Domain names point to *destinations* – other domain names with either the .tor or .onion TLD.

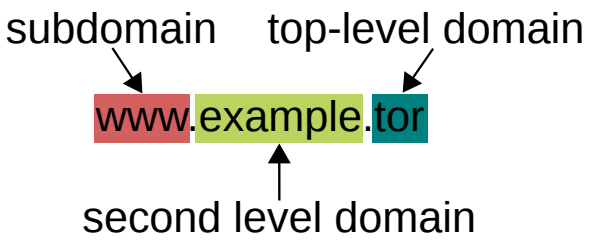

Figure 5.1: A sample domain name: a sequence of labels separated by delimiters. In OnioNS, hidden service operators build associations between second-level domain names and their hidden service address.

The Internet DNS defines a hierarchy of administrative realms that are closely tied to the depth of each name. By contrast, OnioNS makes no such distinction; we let hidden service operators claim second-level names and then control all names of greater depth under that second-level name.

Record

A *Record* is a small textual data structure that contains one or more domain-destination

pairs, proof-of-work, a digital signature, and a public key. Records are issued by hidden service operators and sent to OnioNS servers. Every Record is self-signed with the hidden service's key. In section 5.5.1 we describe the five different types of Records: Create, Modify, Move, Renew, and Delete. A Create Record represents a registration on an unclaimed second-level domain name, Modify, Move, and Renew are operations on that domain name or its subdomains, and a Delete Record relinquishes ownership over the second-level domain name and all its subdomains.

broadcast

The term for the protocol described in section 5.6.1 that describes the uploading of new Records to OnioNS servers through Tor circuits and then the subsequent confirmation that the Record has been received.

Snapshot

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

Page

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

Mirror

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

Quorum Candidate

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

Quorum

A *Quorum* is a subset of Quorum Candidates who have active responsibility over maintaining the master OnioNS Pagechain. Each Quorum node actively its own Page, which has a lifetime of that Quorum. The Quorum is randomly chosen from Quorum Candidates as described in section 5.6.3.

flood

The periodic burst of communication that occurs between Quorum nodes wherein Snapshots are exchanged and each Quorum node obtains the state and digital signature of the current Page maintained by all other Quorum nodes. This communication is described in section 5.6.2.

Throughout the rest of this document, let Alice be a Tor client, and Bob be the operator of a hidden service. Assume that neither Alice nor Bob run Tor relays themselves, but that they only use Tor. Therefore Alice and Bob can obtain and fully verify any past or current set of the three consensus documents described in section 1.2.2. Bob has access to his hidden service private RSA key and may or may not have a PGP key with a subkey for email encryption.

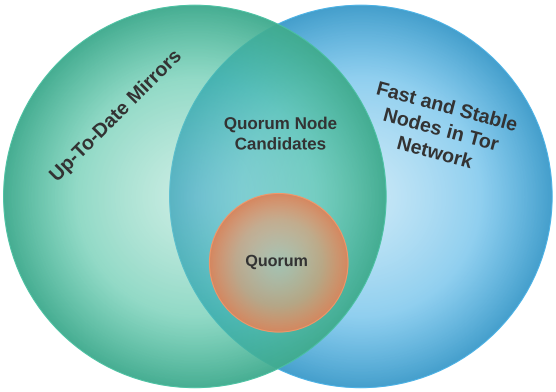

Figure 5.2: The relationship between Mirrors, Quorum Candidates, and the Quorum. A Mirror is any machine that holds the OnioNS Pagechain, Quorum Candidates are both up-to-date Mirrors and reliable Tor nodes, and the Quorum is randomly selected from the pool of Quorum Candidates.

5.3 Basic Design

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

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

These steps describe the propagation of any Record throughout the entire network. The protocols for each party are described in sections 5.6.1, 5.6.2, and 5.6.3, respectively. It should be noted that the activities of Alice, Bob, and the Quorum occur asynchronously: hidden service operator may be transmitting Records to the Quorum at the same time that clients are querying for other domain names.

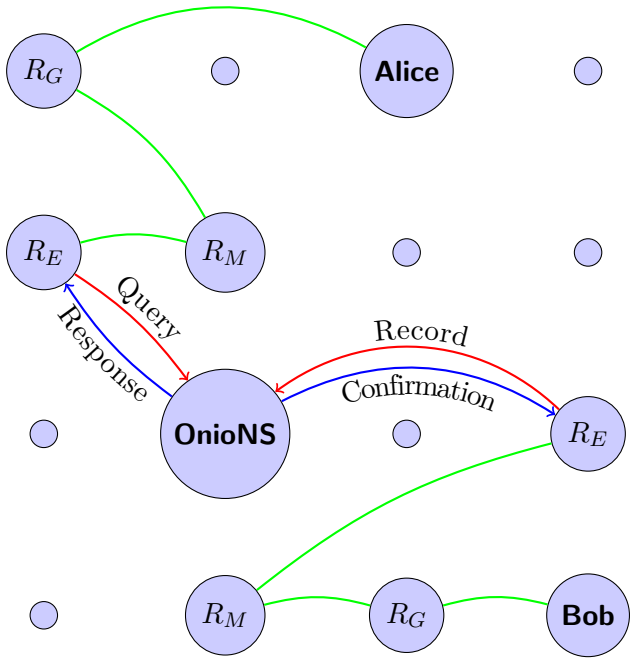

Figure 5.3: Bob uses a Tor circuit (guard, middle, and exit Tor routers) to anonymously broadcast a record to OnioNS. Alice uses her own Tor circuit to query the system for a domain name, and she is given Bob's record in response. Then Alice connects to Bob by Tor's hidden service protocol (section 1.2.3).

5.3.1 Domain Name Operations

Bob first generates a valid Record which he transmits over a Tor circuit to a randomly-selected Quorum node, who confirms acceptance or returns an error message otherwise. Bob's Record is broadcasted to the rest of Quorum via the flood protocol. Bob can confirm that the rest of the Quorum received his Record by constructing another circuit to a different randomly-selected Quorum node and asking it for his Record. This process is illustrated in Figure 5.4.

5.3.2 Pagechain Maintenance

The Pagechain is OnioNS' fundamental data structure. It is a distributed append-only transactional database of a fixed maximum length. The Pagechain is designed as a public and fully confirmable data structure – that is, any Mirror can check the integrity of each Page and the Records contained within them, the uniqueness of domain names, and the

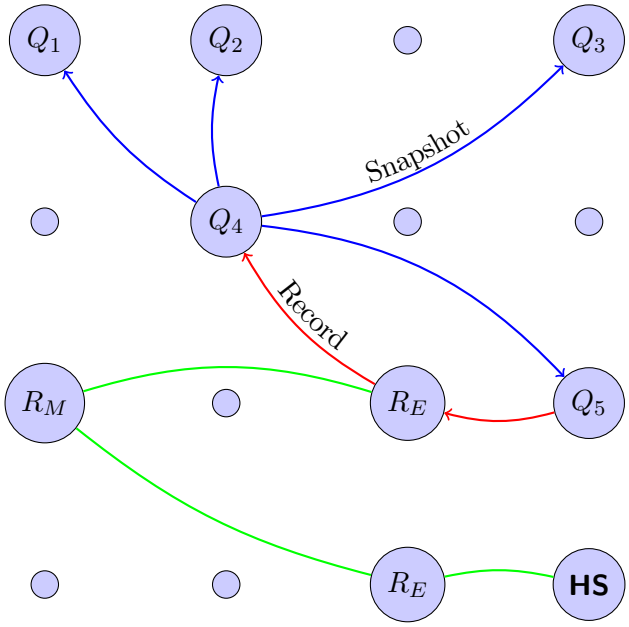

Figure 5.4: Bob uses his existing circuit (green) to inform Quorum node Q_4 of the new record. Q_4 then floods it via Snapshots to all other Quorum nodes. Each node stores it in their own Page for long-term storage. Bob confirms from another Quorum node Q_5 that his Record has been received.

validity of the digital signatures on each Page from all Quorum nodes. The head of the chain (the latest Page) is maintained and digitally signed by each member of the current Quorum. When a Quorum Candidate is chosen as a Quorum node, it generates a new Page and adds that Page onto the front of its Pagechain. As the Quorum receives new Records, Mirror synchronize against them and hold a copy of their latest Page.

Each Quorum node shares the Records that it receives with other Quorum nodes. Assuming that the entire Quorum is honest and maintains perfect communication, all Quorum nodes would be maintaining an identical Page. However, a Quorum node may miss the flood, may act maliciously, or may deviate from the rest of the Quorum for other reasons. Therefore, disagreements may inevitably form within the Quorum. In Section 2.1 we assumed that the largest set of agreeing Quorum nodes is also honest, therefore all Mirrors follow this rule to select the master Pagechain. This assumption is also made when creating a new Page, as described in section 5.6.2. Thus the Pagechain is at least partially self-healing. We illustrate this in Figure 5.5.

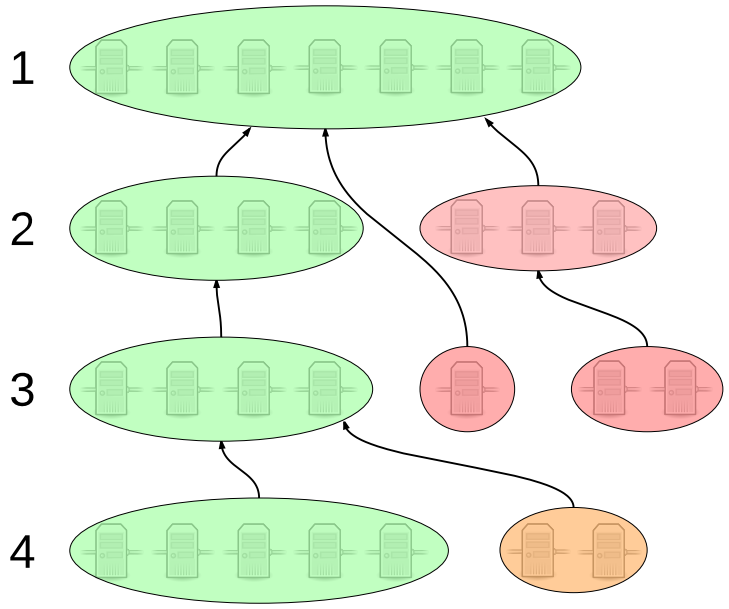

Figure 5.5: An example Pagechain across four Quorums with three side-chains. Quorum_1 is honest and maintains reliable flooding communication, and thus has identical Pages. Quorum_2's largest cluster is likewise in agreement, but there are three nodes which are acting maliciously together and have changed their Pages. Node 5 in Quorum_3 references an old page in attempt to bypass Quorum_2's records, and nodes 6-7 are colluding with nodes 5-7 from Quorum_2. Finally, Quorum_4 has two nodes that acted honestly but did not record new records, so their Pagechains differ from the others. However, across all four days the largest clusters are honest nodes and thus integrity remains in the master Pagechain.

5.3.3 Client Request

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

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

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

In this way, Alice can query an OnioNS resolver for some Dave-selected domain name and recursively resolve it to a .onion address transparently. The number of resolutions Alice makes is fixed to a maximal length (section 5.5.1) so this algorithm always completes. We discuss minor optimizations and security enhancements to this procedure in section 5.6.3.

5.4 Primitives

5.4.1 Cryptographic

OnionNS makes use of cryptographic hash algorithms, digital signatures, proof-of-work, and a pseudorandom number generator. We require that Tor routers generate an Ed25519 keypair and distribute the public key via the consensus document.

- Let $H(x)$ be a cryptographic hash function. In our reference implementation we define $H(x)$ as SHA-384, a truncated and slight modificated derivative of SHA-512. Currently the best preimage attack of SHA-512 breaks 57 out of 80 rounds in 2^{511} time. [27] SHA-384 is included in the National Security Agency's Suite B Cryptography for protecting information classified up to Top Secret.

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

- Let $S_{ed}(m, e)$ be an Ed25519 digital signature function that accepts a message m and a private key e and returns a 64-byte digital signature. Ed25519 is digital signature scheme designed with high performance, strong implementation security, and minimal keypair and signature sizes objectives in mind. The Ed25519 elliptic curve is birationally equivalent to Curve25519. [28] 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 the signature is valid. Let $V_{ed}(m, E)$ use $H(x)$ as a digest function on m in all use cases.

- Let PoW(i) be a proof-of-work scheme such as a strong key derivation function that accepts an input key k and returns a derived key. Our reference implementation uses a fixed salt and scrypt, a password-based key derivation function which is notable for its large memory and CPU requirements during its operation. The scrypt function provides significantly greater resistance to custom hardware attacks and massively parallel computation primarily due to its memory requirements. This limits attackers to the same software implementation and asymptotic cost as legitimate users. [29] [30] We choose scrypt because of these advantages over other key derivation functions such as SHA-256 or PBKDF2. For these reasons scrypt is also common for proof-of-work purposes in some cryptocurrencies such as Litecoin.

- Let $R(s)$ be a pseudorandom number generator that accepts an initial seed s and returns a list of numerical pseudorandom numbers. We suggest MT19937, commonly known as the Mersenne Twister. This generator is widely used throughout most programming languages and is well known for its speed, long period, and the high quality of its pseudorandom output, although it is not cryptographically secure. [31]

5.4.2 Symbols

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

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

5.5 Data Structures

5.5.1 Record

There are five different types of Records: Create, Modify, Move, Renew, and Delete. The latter four Records mimic the format of the Create Record with minor exceptions. In each case, *type* is set to the Record type.

Create

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

Field	**Required?**	**Description**
type	Yes	A textual label containing the type of Record. In this case, *type* is set to "Create".
nameList	Yes	An array list of domain names and their destinations. The list includes one or more second-level domain names, while the remainder of the list contains zero or more sub-domains (names at level < 2) under the second-level domains. In this way, records can be referenced by their unique second-level domain names. Destinations use either .tor or .onion TLDs.
contact	No	Bob's PGP key fingerprint if he has one. Client can use this to contact Bob over encrypted email.
timestamp	Yes	The UNIX timestamp of when the Record was created.
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 $S_d(m, r)$ where $m = nameList \| timestamp \| consensusHash \| nonce \| pow$ and r is the hidden service's private RSA key.
pubHSKey	Yes	Bob's public RSA key.

Table 5.1: A Create record, which contains fields common to all records. Every record is self-signed and must have verifiable proof-of-work before it is considered valid.

Modify

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

Move

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

Renew

Second-level domain names (and their associated domain names) expire every $L_P \Delta q$ 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 domain names. No modifications to existing domain names can be made in Renew records, and the domain names contained within must already exist in the page-chain. Similar to the Modify and Move records, Renew records have a difficulty of $\frac{\text{difficulty}_{\text{Create}}}{4}$.

Delete

A Delete Record is used to relinquish ownership rights over all domain names under the service's RSA key. 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 Snapshot

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

originTime

Unix time when the snapshot was first created.

recentRecords

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

fingerprint

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

snapshotSig

The output of $S_{ed}(originTime \parallel recentRecords \parallel fingerprint], e)$ where e is the router's private Ed25519 key.

5.5.3 Page

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

prevHash

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

recordList

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

consensusDocHash

The SHA-384 of *cd*.

fingerprint

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

pageSig

The output of $S_{ed}(H(prevHash \parallel recordList \parallel consensusDocHash)], e)$ 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 LempelZivMarkov 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 constructs the *nameList* domain-destination associations.
3. Bob provides his PGP key fingerprint in *contact* or leaves it blank if he doesn't have a PGP key or if he chooses not to disclose it. Bob can derive his PGP fingerprint with the "gpg –fingerprint" Unix command.
4. Bob records the number of seconds since the Unix epoch in *timestamp*.
5. Bob sets *consensusHash* to the output of $H(x)$, where x is the consensus 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*.
8. Bob sets *pow* as PoW(*central*).
9. Bob sets *recordSig* as the output of $S_d(m, r)$ where $m = \textit{central} \,\|\, \textit{pow}$ and r is Bob's private RSA key.
10. Bob saves the PKCS.1 DER encoding of his RSA public key in *pubHSKey*.

Bob then must increment *nonce* and reset *pow* and *recordSig* until $H(\textit{central} \,\|\, \textit{pow} \,\|\, \textit{recordSig}) \leq 2^{\textit{difficulty} * \textit{count}}$ where *difficulty* is a fixed constant that specifies the work difficulty and *count* is the number of second-level domain names claimed in the Record. An example of a completed and valid record is shown in Figure 5.6.

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

Figure 5.6: A sample registration record. The textual fields are in UTF-8, while the binary fields are in base64. The structure is encoded in JSON.

Record Validation

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

1. Carol checks that the Record contains valid JSON.

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

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

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

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

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

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

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

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

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

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

Record Broadcast

1. Bob derives the current Quorum by the Quorum Derivation protocol described in section 5.6.3.

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

3. Bob asks for and receives from c_1 the $h_j = H(prevHash \| recordList \| consensusDocHash)$ 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 waits up to Δs minutes for the next s flood iteration.
8. Bob uses c_1 to ask m_1 for any second-level domain name contained in his Record.
9. Bob has confirmation that his Record was accepted and processed by the Quorum if m_1 returns the Record he uploaded.
10. If m_1 does not return Bob's Record, he can either query another Mirror or repeat this procedure and broadcast to a different Quorum node.

5.6.2 OnioNS Servers

Let Charlie be the name of an OnioNS Mirror.

Database Initialization

1. Charlie creates a empty Snapshot.
2. Charlie sets *originTime* to the number of seconds since the Unix epoch.
3. Charlie sets *recentRecords* to an empty array.
4. Charlie sets *fingerprint* to his Tor fingerprint.
5. Charlie sets $snapshotSig = S_{ed}(originTime \parallel recentRecords \parallel fingerprint, e)$ where e is Charlie's private Ed25519 key.
6. Charlie begins listening for incoming Records.
7. Charlie creates an initial Page P_{curr}.
8. Charlie selects a previous Page P_{prev} to back-reference via the Page Selection protocol.
9. Charlie sets $prevHash = H(P_{prev}(prevHash \parallel recordList \parallel consensusDocHash)$.
10. Charlie sets *recordList* to an empty array.
11. Charlie downloads from some remote source the consensus document *cd* issued on day $\lfloor \frac{q}{\Delta q} \rfloor$ at 00:00 GMT and authenticates it against the Tor authority public keys.

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

Figure 5.7: A sample empty Page.

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

Figure 5.8: A sample empty Snapshot.

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

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

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

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 obtains the set of Pages maintained by $Quorum_q$.
2. Charlie obtains the consensus documents cd issued on day $\lfloor \frac{q}{\Delta q} \rfloor$ at 00:00 GMT and authenticates them.
3. Charlie uses cd to calculate the Quorum via the Quorum Derivation protocol.
4. For each Page,
   (a) Charlie checks that *prevHash* references some page from the previous Quorum.
   (b) Charlie calculates $h = H(prevHash \parallel recordList \parallel consensusDocHash)$.
   (c) Charlie checks that *fingerprint* is a member of $Quorum_q$.
   (d) Charlie checks that $V_{ed}(h, E)$ returns true.
5. Charlie sorts the set of Pages by h, and constructs a 2D array of Pages that have the same h.
6. For each set of Pages with equal h,
   (a) Charlie checks that $consensusDocHash = H(cd)$.
   (b) Charlie checks each Record in *recordList* via the Record Validation protocol.
7. If the validation of a Page fails, Charlie removes it from the equal-h list.
8. Let P_c be chosen arbitrarily from the largest set of valid Pages with equal h.

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

Pagechain Validation

Assume that Charlie has obtained a complete Pagechain. Let P_c be an initial empty Page and let $f(P_1, P_2, q)$ accept two Pages and return true if $P_1 = P_2$ or if $q = 0$. For each q_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 $h_i = H(prevHash \parallel recordList \parallel consensusDocHash)$, 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 Quorum Candidate, R_j.
2. Charlie downloads R_j's Pagechain and the Pages used by each Quorum member for each Quorum, obtaining a 2D data structure at most L_P Pages long and L_Q Pages wide.
3. Charlie checks the downloaded Pagechain via the Pagechain Validation protocol. If it does not validate, Charlie picks another Quorum Candidate R_k, k/nej, and downloads the invalid Pages from R_k.
4. Charlie randomly selects a Quorum node Q_c.
5. Charlie periodically polls Q_c for the Pages of all other Quorum nodes.
6. When a set of Pages is available from Q_c, Charlie follows the Page Selection protocol to choose a Page that becomes the new Pagechain head.

Quorum Qualification

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

The first criteria requires Tor nodes to demonstrate that they sufficient capabilities to handle the increase in communication and processing from with OnioNS protocols. Fortunately, Tor's infrastructure already provides a mechanism that can be utilized to demonstrate this requirement; Tor authority nodes assign flags to Tor routers to classify their capabilities, speed, or uptime history: these flags are used for circuit generation and hidden service infrastructure. Let Tor nodes meet the first qualification requirement if they have the Fast, Stable, Running, and Valid flags. As of February 2015, out of the $\approx$ 7,000 nodes participating in the Tor network, $\approx$ 5,400 of these node have these flags and complete the second requirement. [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 minute component of the time of day in GMT. Tor's consensus documents are published at the top of each hour; we manipulate m such that t is consistent at the top of each hour even with at most a 15-minute clock-skew.

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

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

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 it does not validate according to the Record Validation protocol.
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 any of its second-level domains already exist 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 names.
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,
   (a) Q_j informs Bob that r was accepted.
   (b) Q_j merges r into its current Snapshot as shown in Figure 5.9.
   (c) Q_j regenerates *snapshotSig*.

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

Figure 5.9: A sample Create Record has been merged into an initially-blank Snapshot.

Flooding

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

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

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

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

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

   (a) Q_j asks Q_k for his snapshot.

   (b) Q_j merges the Records within Q_k's *recentRecords* into Q_j's current Page, as shown in Figure 5.10.

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

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

5. Q_j regenerates *pageSig*.

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

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

5.6.3 Tor Clients

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

Quorum Derivation

1. Alice obtains 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 $f = R(H(cd)$.

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

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

4. Alice uses f to randomly scramble qc.

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

Domain Query

There are three verification levels in a Domain Query, each providing progressively more verification to Alice that the Record she receives is authentic, unique, and trustworthy. At verification 0, the default:

1. Alice constructs a Tor circuit to Charlie.
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 at level 0.
5. Charlie reviews his Pagechain reverse-chronological order until he finds a Record containing d, which he returns to Alice.
6. Alice validates r via the Record Validation protocol. If it does not validate, she changes to another resolver and repeats this protocol.
7. If the destination for d in r uses a .tor TLD, d becomes that destination and Alice jumps back to step 3.
8. Otherwise, the destination must have a .onion TLD, which Alice looks up by the Tor hidden service protocol.
9. Alice checks that r's *pubHSKey* matches r's key in Tor's distributed hash table.
10. Alice sends the original d to the hidden service.

At verification level 1, Charlie returns returns the Page p containing r in step 5. Then Alice can check the integrity of p via the Page Selection protocol and can verify its authenticity via checking that $V_{ed}(prevHash \parallel recordList \parallel consensusDocHash, E)$ returns true.

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

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 MiB/s, [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 that .onion as a destination and returns the results corresponding to the verification level. Onion Queries have a high likelihood of failure as not every .onion name has a corresponding .tor domain name, but all OnioNS domain names will have Forward-Confirmed Reverse DNS match.

5.7 Optimizations

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

5.7.1 AVL Tree

An AVL tree is a self-balancing binary search tree with $\mathcal{O}(\log(n))$ time for search, 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 constructs a list of associations between a second-level domain name and the location in the Pagechain of the Record containing that domain. The Mirror then inserts that list into the AVL tree, where sorting occurs by alphabetical comparison of domain names. 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 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 58 and a maximum depth of 16. Let the leaves of the trie be the location of the most recent Record in the Pagechain that has that address as a destination. Like the AVL tree, Mirrors must take care to update the trie cache when processing new Records, but this is efficient as trie search, insertion, and deletion all occur in $\mathcal{O}(1)$.

5.7.3 Hashtable Bitset

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

As an extension of an ordinary hashtable, the hashtable bitset maps keys to buckets, but here we only track the existence of a key but not the keys themselves. Therefore each bucket in the structure is represented as a bit, creating a compact $z * n$-length bitset, where

z is a scaling factor. Let the hashtable use the $H(m)$ function from section 5.4.1. A Mirror constructs a hashtable bitset in $\mathcal{O}(n)$ time by the following algorithm:

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

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

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

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

CHAPTER 6

ANALYSIS

6.1 Security

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

6.1.1 Quorum Selection

The Quorum nodes have greater attack capabilities than any other class of participants in OnioNS. They have active responsibility over the front of the Pagechain and they must receive, flood, and process new Records from hidden service operators. In our threat model, we assume that an attacker, Eve, already has control of some fixed number f_E of routers in the Tor network, and that her nodes may maliciously collude. We also assume that Eve does does not have motivation to compromise Tor in response to the presence of OnioNS and that she cannot predict future Quorums. 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 assumptions.

Our security analysis assumes that L_Q will be selected from a pool of 5,400 Quorum Candidates — the number, as of April 2015, of Tor routers with the Fast and Stable flags,

whom we assume are all up-to-date Mirrors. Let L_E be the number of Quorum nodes under Eve's control. Then Eve controls the Quorum if the L_E routers become the largest agreeing subset in the Quorum, which can occur if either more than $\frac{L_Q - L_E}{2}$ honest Quorum nodes disagree or if $L_E > \frac{L_Q}{2}$. 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.

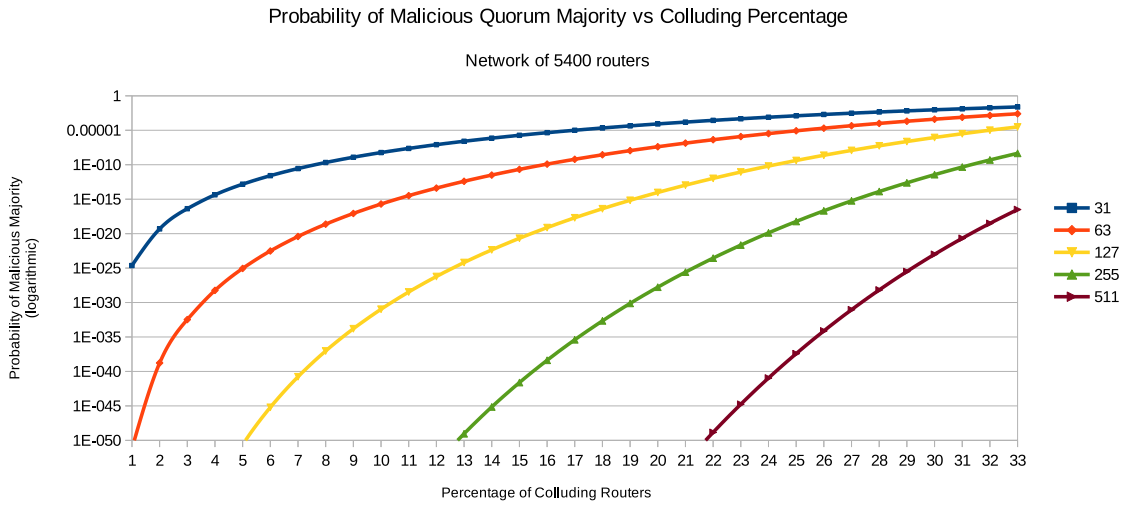

Figure 6.1: The probability that Eve controls the majority of the Quorum is given by the PMF of the hypergeometric distribution. We fix N at 5,400 nodes and graph Eve's success probability as a function of an increasing percentage of Eve-controlled colluding routers. We examine five selections for L_Q: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

Figure 6.1 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.

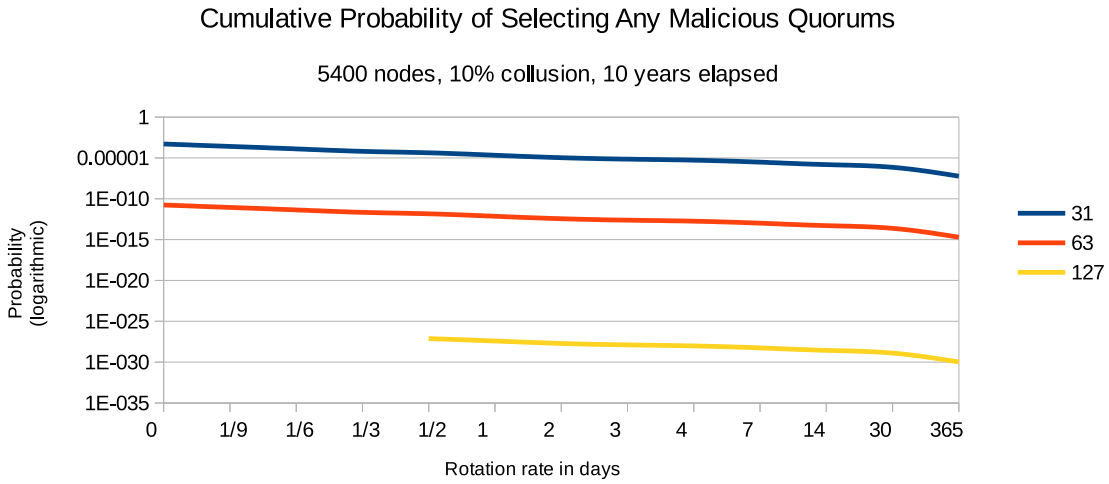

Figure 6.2: The cumulative probability that Eve controls any Quorum at different rotation rates. We assume 10 percent collusion in a network of 5400 Tor routers, and view across 10 years. We do not graph L_Q values of 255 or 511 as they generate probabilities far below our $10^{-38.532}$ threshold; $L_Q = 255$ 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, as shown in Figure 6.3. This figure suggests that fast rotations are optimal in that respect, a contradiction to Figure 6.1. However, given the very low statistical likelihood of selecting a malicious Quorum, we consider this a minor contribution to the decision.

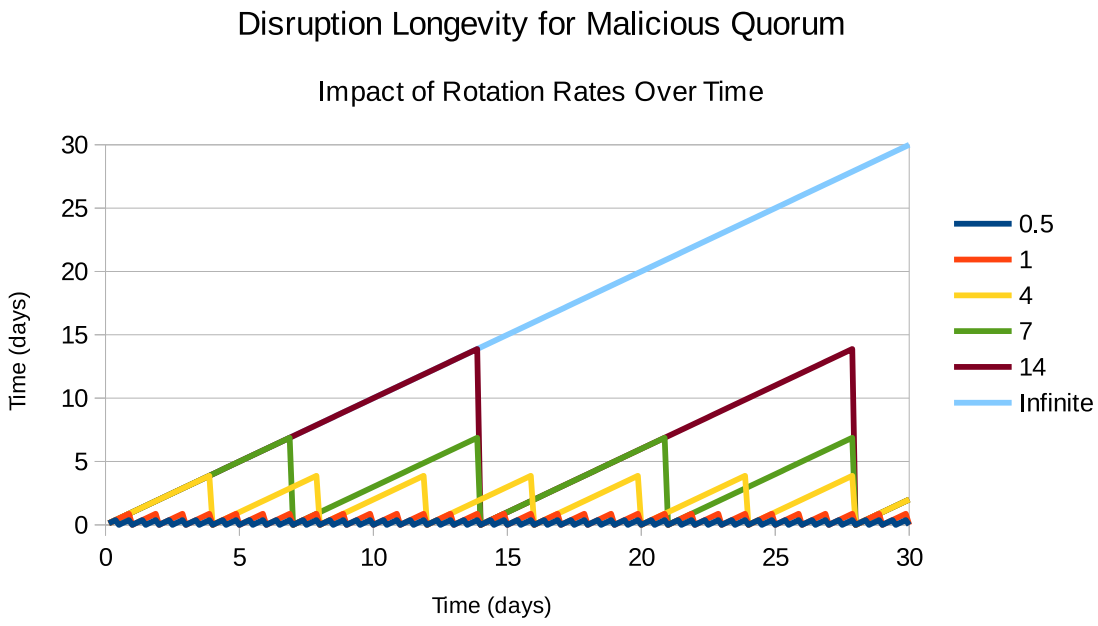

Figure 6.3: The duration of malicious Quorums as a function of different rotation rates. Quorums that have very short livetimes (and are thus rotated quickly) minimize the duration of any malicious activity.

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

In section 1.2.2 we detailed the significant contents of the three consensus documents relevant to OnioNS, *cached-certs*, *cached-microdesc-consensus*, and *cached-microdescs*. As we stated in section 5.6, the Quorum Derivation protocol only utilizes the *cached-certs* and *cached-microdesc-consensus* documents for reasons we discuss below.

cached-certs

The *cached-certs* document contains long-term directory identity keys and medium-term signing keys. While Eve cannot predict the public half of new signing keys in advance, the keys rotate every 3-12 months [16] so *cached-certs* is not a timely source of entropy. Nevertheless, if an attacker can predict the network status described in *cached-microdesc-consensus*, the rotation timeline of the signing keys in *cached-certs* places an absolute upper-bound on the duration of Quorum predictability.

cached-microdesc-consensus

The *cached-microdesc-consensus* document describes the network status and is our main source of entropy. In the header, the *vote-digest* header is the hash of a directory authority's status vote. Each directory operates independently and there are nine directory authorities, so we consider this value unpredictable. Router descriptors follow the header in the body of the document.

The r field in a router's descriptor contains routing information and time of last restart. Tor routers have no guarantee of availability and routers may restart for a variety of reasons. As of April 2015 Tor's network consists of approximately 7,000 routers [1] and assuming that a given router restarts every 60 days, statistically the number of unpredictable r fields each day is given by $\frac{7000}{60} \approx 116.66$. Tor displays the restart time down to the second, so each restart adds approximately six bytes of entropy, for an estimated total of $\frac{7000*6}{60} = 700$ bytes of short-term entropy from the r field.

The v field describes the version of Tor that the router is running. Although the versions of Tor are publicly known and the official and unofficial Linux repositories are publicly accessible, Eve cannot predict when an administrator will upgrade their router to the next available Tor version. Although new versions of Tor are not frequently released and routers are upgraded infrequently as shown in Figure 6.4, the v field still introduces a degree of medium-term entropy into the document.

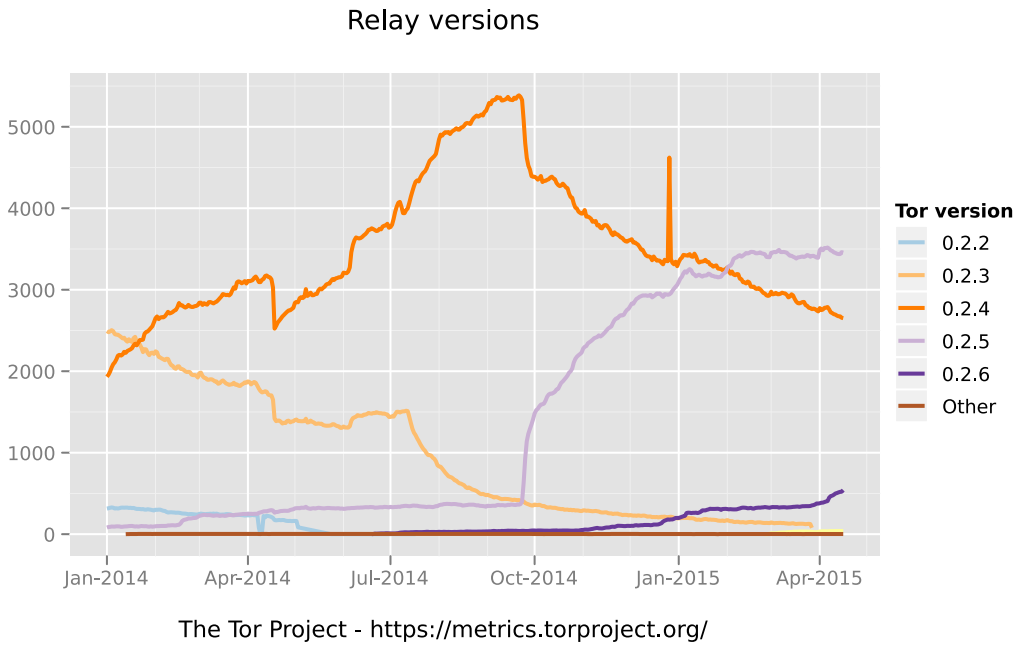

Figure 6.4: The versions of Tor software between January 2014 and April 2015. [1]

The w field contains the router's estimated bandwidth capacity as calculated by bandwidth authorities. Clients use this field during circuit construction; routers with a higher bandwidth capacity relative to the rest of the network have a higher probability of being included in a circuit. Although it is likely that a given router will have similar bandwidth measurements between consecutive consensus documents, Eve cannot predict the exact performance of a router from the perspective of the bandwidth authority. Eve's capacity to predict the performance of all routers falls outside of Tor's and OnioNS' security assumptions; Eve must therefore be a global attacker who would be capable of compromising the Tor network as a whole anyway.

We note that significant amounts of additional entropy could be trivially added into *cached-microdesc-consensus* if each router added a single random byte to their descriptor or if the directory authorities each contributed entropy.

cached-microdescs

Although we did not detail it in section [16], in practice Tor places client-side timestamps inside *cached-microdescs*. These timestamps would significantly divide the network, causing *cached-microdescs* to be ill-suited for inclusion in the Quorum Derivation protocol. Although *cached-microdescs* contains long-term router keys and the fingerprints of routers in the same family which would both serve as a source of long-term entropy, *cached-microdesc-consensus* contains the SHA-256 hashes of the router descriptor. We therefore do not include it when hashing the documents in the Quorum Derivation protocol.

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) then Eve can find k seeds that generates a desirable scrambled list in 2^{192} operations on average, or 2^{384} operations in the worst case. The chance of any of those seeds being selected is $\frac{2^{384}}{k}$.

If we also assume that Eve can predict some fraction $f \in (0, 1)$ of the contents of the consensus documents, then Eve may attempt to manipulate her router's descriptors such that the Quorum Derivation protocol produces one of the k hashes. SHA-384's strong resistance to preimage and second preimage attacks requires 2^{192} operations on average for Eve to find one of the k hashes. This difficulty is compounded by the limited combinatorics within the set of valid router descriptors. 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. We do not consider the possibility of Eve controlling the entire consensus document, this would severely compromise the integrity of the Tor network and thus violates our security assumptions.

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*, *nameList*, *contact*, *timestamp*, and *consensusHash* fields.
2. Bob sends R to Craig.
3. Let *central* be *type* $\|$ *nameList* $\|$ *contact* $\|$ *timestamp* $\|$ *consensusHash* $\|$ *nonce*.
4. Craig generates a random integer K and then for each iteration j from 0 to K,
   (a) Craig increments *nonce*.

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

(c) Craig saves the new R as C_j.

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

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

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

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

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

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

6.1.6 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. [32] 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:

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

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

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

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

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

6. **The system must be simple and relatively easy to use** — Domain Queries are automatically resolved and require no input by the user. From the user's perspective, they are taken directly from a meaningful domain name to a hidden service. Users no longer have to use unwieldy .onion addresses or review third-party directories, OnioNS introduces memorability to hidden service domains.

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

Finally, we meet our optional performance objectives:

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

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

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

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

CHAPTER 7

IMPLEMENTATION

7.1 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 at `https://github.com/Jesse-V/OnioNS`.

7.2 Prototype Design

We have developed an OnioNS prototype that implements the Domain Query protocol. In this initial prototype, we use a static server, a single fixed Create Record, and a hidden service that we deployed at onions55e7yam27n.onion. Although our implementation is primarily a separate software package, we made necessary modifications to the Tor client software to intercept the .tor pseudo-TLD, pass the domain to the OnioNS client over interprocess communication (IPC), and receive and lookup the returned hidden service address. We diagram the relationships between our prototype's components in Figure 7.1.

Our prototype's works as follows:

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 communicates with Tor's SOCKS port and negotiates a connection to the static OnioNS server.

5. The OnioNS client performs a level 0 Domain Query to the server.

6. The server responds with the Create Record containing "example.tor".

7. The client writes "onions55e7yam27n.onion" to the Tor client over another named pipe.

8. The Tor client resumes the lookup and overrides the original "example.tor" lookup with "onions55e7yam27n.onion".

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

10. The Tor Browser displays the website contents and preserves the "example.tor" domain name.

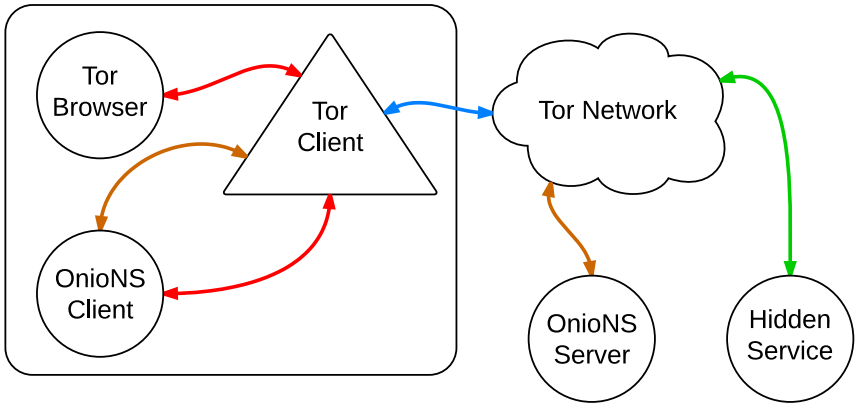

Figure 7.1: Our OnioNS prototype involves five components: the Tor Browser, a modified Tor client, a OnioNS client, a OnioNS server, and the destination hidden service. The Tor Browser passes an unknown .tor domain to the OnioNS through the Tor software (shown in red) which resolves the domain anonymously over a Tor circuit (orange) to remote resolver. Finally, the Tor software contacts the destination server via the normal hidden service protocol (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).

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 created a rudimentary form of the Record Generation and Client Verification protocols and conducted performance measurements on two different machines. We tested these protocols on Machine A, which has an Intel Core2 Quad Q9000 @ 2.00 GHz; and machine B, which has an Intel i7-2600K @ 4.3 GHz. We chose machines with this hardware in order to more accurately determine the performance with out target audience; Machines A and B 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. As the Record Generation protocol requires a hidden service private key,

we created a hidden service and hosted it on Machine B, using Shallot, a vanity key generator, to generate a recognizable hidden service address, `http://onions55e7yam27n.onion`. 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. This is an affordable amount of RAM by even low-end consumer-grade machines. This RAM consumption scales linearly with the number of scrypt instances executed in parallel. We used all eight cores on Machine B to generate a Record for our hidden service and observed approximately 1 GB of RAM consumption, matching our expectations. 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.

7.2.3 Future Work

We have several plans for future work. First, we will port the OnioNS client to Windows and migrate the Tor-OnioNS IPC from Unix named pipes to TCP sockets. Although named pipes work reliably in Unix-like operating systems, they are not easily compatible in the Windows family of operating systems. Second, we will find more optimal parameters for scrypt to increase performance and work to reduce the latency. Finally, we will expand our implementation and collaborate with Tor developers to merge it into Tor's repositories when it is complete.

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 *Proceedings of the 27th Annual Computer Security Applications Conference*. 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 *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. IEEE, 1997, pp. 44–54.

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

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

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

[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," 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," 2009.

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

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

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

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

[28] 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.

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

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

[31] 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.

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