Anonymized Submission

The Onion Name System

Tor-powered Decentralized DNS for Tor Onion Services

Abstract: Tor onion services, also known as hidden services, are anonymous servers of unknown location and ownership that can be accessed through any Tor-enabled client. They have gained popularity over the years, but since their introduction in 2002 still suffer from major usability challenges primarily 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 decentralized resolution service. OnioNS allows Tor users to reference an onion service by a meaningful globally-unique verifiable domain name chosen by the onion service administrator. We construct OnioNS as an optional backwards-compatible plugin for Tor, 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. We introduce a lottery-like system to reduce the threat of land rushes and domain squatting. Finally, we provide a security analysis, integrate our software with the Tor Browser, and conduct performance tests of our prototype.

1 Introduction

As the prevalence of the Internet and other communication has grown, so too has the development and usage of privacy-enhancing systems. These are protocols that provide privacy by obfuscating the link between a user's identity or location and their communications. Following a general distrust of unsecured Internet communications and in light of the 2013-current revelations by Edward Snowden of international Internet mass-surveillance, users have increasingly turned to these tools for their own protection.

Tor [7] is a third-generation onion routing system and is the most popular low-latency anonymous communication network in use today. In Tor, users construct a layered encrypted communications circuit over three onion routers in order to mask their identity and location. As messages travel through the circuit, each onion router in turn decrypts their encryption layer, exposing their respective routing information. The first router is only exposed to the user's IP address, while the last router conducts Internet activities on the user's behalf. This provides end-to-end communication confidentiality of the sender.

Tor users interact with the Internet and other systems over Tor via the Tor Browser, a security-enhanced fork of Firefox ESR. This achieves a level of usability but also security: Tor achieves most of its application-level sanitization via privacy filters in the Tor Browser; unlike its predecessors, Tor performs little sanitization itself. Tor's threat model assumes that the capabilities of adversaries are limited to traffic analysis attacks on a restricted scale; they may observe or manipulate portions of Tor traffic, that they may run onion routers themselves, and that they may compromise a fraction of other existing routers. Tor's design centers around usability and defends against these types of attacks.

1.1 Motivation

Tor also supports *onion services* – anonymous servers that intentionally mask their IP address through Tor circuits. They utilize the .onion pseudo-TLD, typically preventing the services from being accessed outside the context of Tor. Onion services are only known by their public RSA key and typically referenced by their address, 16 base32-encoded characters derived from the SHA-1 hash of the server's key, i.e. 3g2upl4pq6kufc4m.onion. This builds a publicly-confirmable one-to-one relationship between the public key and its address and allows onion services to be accessed via the Tor Browser by their onion address within a distributed environment.

Tor onion addresses are decentralized and globally collision-free, but there is a strong discontinuity between the address and the service's purpose. As their addresses usually contain no human-readable information, a visitor cannot categorize, label, or authenticate onion ser-

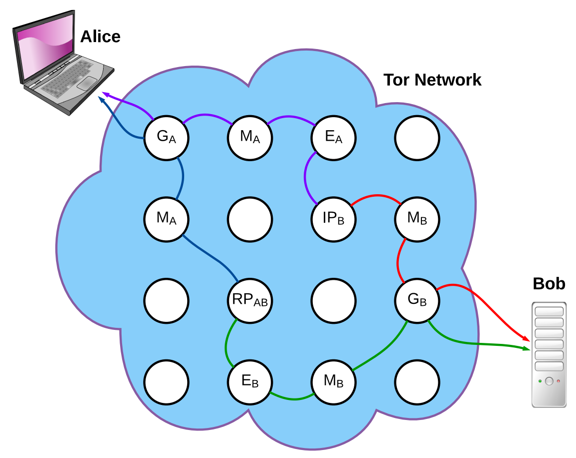

Fig. 1. A Tor client, Alice, and an onion service, Bob, first mate two Tor circuits (purple and red) at one of Bob's long-term *introduction points* (IP). They then renegotiate and communicate over another pair of Tor circuits (blue and green) at an ephemeral *rendezvous point* (RP). This achieves communication with bi-directional anonymity [18].

vices in advance. While a Tor user may explore and bookmark onionsites within the Tor Browser, this is a very narrow solution and does not scale well past a few dozen bookmarks. Over time, third-party directories – both on the clearnet and onionspace – have appeared in an attempt to counteract this issue, but these directories must be constantly maintained and the approach is neither convenient nor does it practically scale past several hundred entries. The approximately 27,000 onion services currently on the Tor network (Figure 2) and the potential for continued growth both suggest the strong need for a more complete and wider solution to solve the usability issue.

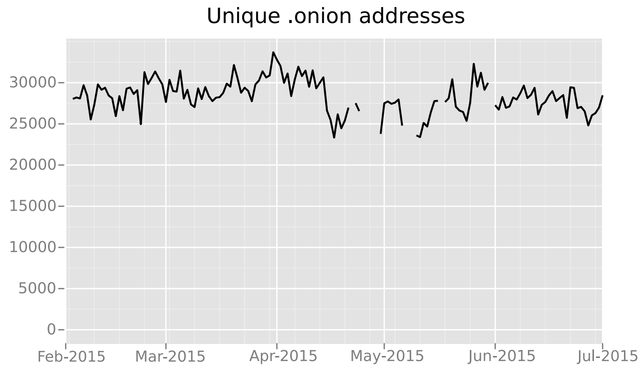

Fig. 2. The number of unique onion addresses seen in the Tor network between February 2015 and July 2015 [11, 22].

1.2 Contributions

In this paper, we present the design, analysis, and implementation of the Onion Name System, (OnioNS) a decentralized, secure, and usable domain name system for Tor onion services. Any onion service administrator can claim a meaningful human-readable domain name without loss of anonymity and clients can query against OnioNS in privacy-enhanced and verifiable manner. OnioNS is powered by a random subset of nodes within the existing Tor network, significantly limiting the additional attack surface. We devise a distributed database that is resistant to node compromise and provides authenticated denial-of-existence. We provide a backwards-compatible plugin for the Tor Browser and demonstrate the low latency and high performance of OnioNS. To the best of our knowledge, this is the first alternative DNS for Tor onion services which is decentralized, secure, and privacy-enhanced. Moreover, no other alternative DNS provides mechanisms for authenticated denial-of-existence.

Paper Organization: This paper is divided into four main sections. In section 2 we define our design objectives and explain why existing works do not meet our goals. We also define our threat model, which includes Tor's assumptions and the capabilities of our adversaries. In section 5, we describe the system overview and define several key protocols. In section 6 we analyse the security of our assumptions and examine other attack vectors. Last, in section 7 we describe our implementation prototype, perform performance analysis tests, and demonstrate that our software allows the Tor Browser to load an onion service under a meaningful domain name.

2 Problem Statement

To integrate with Tor, we must provide a secure system, preserve user privacy, and avoid compromising other areas of the Tor network. Additionally, we seek to achieve all three properties of Zooko's Triangle (section **??**) and to providing a mechanism for authenticated denial-of-existence (section **??**).

2.1 Design Objectives

Tor's privacy-enhanced environment introduces distinct challenges to any new infrastructure. Here we enumerate a list of requirements that must be met by any naming system applicable to Tor onion services. In Section 3 we analyze existing works and show how these systems do not meet these goals and in Section 5 we demonstrate how we overcome them with OnioNS.

1. **Anonymous registrations**: The system should not require any personally-identifiable or location information from the registrant. Tor onion services publicize no more information than a public key and a set of Introduction Points.

2. **Privacy-enhanced queries**: Clients should be anonymous, indistinguishable, and unable to be tracked by name servers. Tor already tunnels most Internet DNS queries over circuits, thus any alternative naming system should continue to preserve user privacy during lookups.

3. **Strong integrity**: 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 onion service. This objective provides a defense against phishing attacks.

4. **Globally unique domain names**: 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. Unique domain names prevent fragmentation of users and also provides a defense against phishing attacks.

5. **Decentralized control**: Central authorities carry absolute control over the system and root security breaches can easily compromise the integrity of the entire system. They may also be able to compromise the privacy of both users and onion services or may not allow anonymous registrations.

6. **Low latency**: The Tor network introduces noticeable latency into communication, especially for onion services, although this is not by design. The system must promptly resolve queries to avoid negatively impacting usability and exhausting the patience of Tor users.

7. **Optional**: Not all onion services require meaningful names. For example, applications such as Ricochet [4] may create ephemeral onion services where names may not be appropriate or necessary. Thus a naming system should be optional but not required. Systems that provide backwards compatibility by preserving the Tor onion service protocol also achieve this property.

8. **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. The system should have a minimal impact on Tor clients and onion services.

3 Related Works

Vanity key generators (e.g. Shallot [12]) attempt to find by brute-force an RSA key that generates a partially-desirable hash. Vanity key generators are commonly used by onion service administrators to improve the recognition of their onion service, particularly for higher-profile services. For example, a onion service administrator may wish to start his service's address with a meaningful noun so that others may more easily recognize it. However, these generators are 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. If the address key-space was reduced to allow a full brute-force, the system would fail to be guaranteed collision-free. Nicolussi suggested changing the address encoding to a delimited series of words, using a dictionary known in advance by all parties [17]. While Nicolussi's encoding improves the readability of an address, like vanity key generators it does not allow addresses to be completely meaningful.

The Internet DNS is already well established as a fundamental abstraction layer for Internet routing. However, despite its widespread use and extreme popularity, DNS suffers from several significant shortcomings and fundamental security issues that make it inappropriate for use by Tor onion services. First, the Internet DNS 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 [26]. Additional protocols such as DNSCurve [2] have been proposed, but DNSCurve has not yet seen widespread deployment across the Internet. Secondly, both DNS and DNSSEC are highly centralized; the entire .com TLD, for example, is under the control of Verisign in the USA. The lack of default security in DNS and its fundamental centralization prevent us from using it for onion services.

OnionDNS [24] is a seizure-resistant alternative resolution service for the Internet. OnionDNS is based on DNS and uses unmodified BIND client software but anonymizes the root server by hosting it as an onion service. While OnionDNS is highly usable and provides DNSSEC and other authentication mechanisms, the system is centralized by a single root server and thus highly vulnerable if the root is malicious or is compromised. OnionDNS and OnioNS were named independently and readers should take care to not confuse the two works.

The GNU Name System [26] (GNS) is a decentralized alternative DNS. GNS distributes names across a hierarchical system of zones constructed into directed graphs. Each user manages their own zone and distributes zone access peer-to-peer within social circles. However, GNS 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 onion services and such a selection centralizes control of the system. Awerbuch and Scheideler, [1] constructed a decentralized peer-to-peer naming system, but like GNS, made no guarantee that domain names would be globally unique.

Namecoin [5] is an early fork of Bitcoin [15] and is the first fully-distributed alternative DNS that distributes meaningful and unique names. Like Bitcoin, Namecoin holds information transactions in a decentralized ledger known as a blockchain. Transactions and information are added to the head of the blockchain by "miners," who solve a proof-of-work problem to generate the next block. While Namecoin is often advertised as capable of assigning names to Tor onion services, it has several practical issues that make it generally infeasible to be used for that purpose. First, Namecoin generally requires users to pre-fetch the blockchain which introduces significant logistical issues due to high bandwidth, storage, and CPU load. Second, since the blockchain is an append-only data structure, it becomes less practical over time and scales poorly to high levels of activity and popularity. Third, although Namecoin supports anonymous ownership of information, it is non-trivial to anonymously purchase Namecoins, thus preventing domain registration from being privacy-enhanced. These issues prevent Namecoin from being a practical alternative DNS for Tor onion services.

4 Assumptions and Threat Model

We assume that Tor circuits provides privacy and anonymity; if Alice constructs a three-hop Tor circuit to Bob with modern Tor cryptographic 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. This also implies an assumption on the security of cryptographic primitives and a lack of backdoors or analogous breaks in cryptographic libraries. The security of Tor circuits is also dependent on the assignment of consensus weight; we assume that the majority of directory authorities are at least semi-honest and that consensus weight is an effective defense against Sybil attacks. The aforementioned assumptions are shared by the Tor network.

We assume that an active attacker, Mallory, controls some percentage of dishonest colluding Tor routers as well as semi-honest routers; however this percentage is small enough to avoid violating our second assumption. We assume a fixed percentage of dishonest and semi-honest routers; namely that the percentage of routers under an Mallory'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 Mallory. However, we allow Mallory to operate and actively MitM any of our non-authoritative name servers.

As onion services require no more than a configured Tor client and a socket listener and are thus cheap to create, we anticipate that actors in our system will perform any of the following use cases:

1. **Create many onion services to register many names.**
2. **Create one onion service to register many names.**
3. **Create many onion services to register a single name.**
4. **Create one onion service to register a single name.**

Scenarios one and two are expected to be performed by adversaries attempting to register all popular names in a "land rush" for financial gain or as a denial-of-registration attack. Scenario three may indicate an attempt by many legitimate actors to claim a highly desirable name, while scenario four is the expected behavior of innocent actors. As onion services are anonymous by nature, it is not straightforward to construct a system that differentiates and selects between a single actor performing the first scenario and many actors performing the fourth scenario. However, as well-known onion services cannot change their public key, we expect innocent actors to follow the fourth use case.

5 Solution

5.1 Cryptographic Primitives

OnioNS utilizes hash functions, digital signature algorithms, a proof-of-work scheme, and a global source of randomness.

- Let $\mathcal{H}(x)$ be a cryptographic hash function. We define $\mathcal{H}(x)$ as SHA-256.
- Let $S_{RSA}(m, r)$ be a RSA digital signature function that accepts a message m and a private RSA key r and returns a digital signature. Let $S_{RSA}(m, r)$ use $\mathcal{H}(x)$ as a digest function on m in all use cases. We define $S_{RSA}(m, r)$ as EMSA4/EMSA-PSS.
- Let $V_{RSA}(m, R)$ validate an RSA digital signature by accepting a message m and a public key R, and return true if and only if the signature is valid.
- Let $\mathrm{PoW}(k)$ be a one-way function that accepts an input key k and returns a deterministic output. While $\mathrm{PoW}(k)$ could ideally be set to memory-hard key derivation function such a scrypt, [19] for performance reasons we define $\mathrm{PoW}(k)$ as $\mathcal{H}(x)$.
- Let $\mathcal{G}(t)$ be a cryptographically-secure generator of random or pseudorandom timestapped numbers. $\mathcal{G}(t)$ deterministically returns a value for time t in the present or past.
- Let $\mathcal{R}(s)$ be a pseudorandom number generator that accepts an initial seed s and returns a list of pseudorandom numbers. In our design, $s = \mathcal{G}(t)$, so $\mathcal{R}(s)$ does not need to be cryptographically secure. 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 [13].

5.2 Definitions

domain name The syntax of OnioNS domain names mirrors the Internet DNS; we use a sequence of name-delimiter pairs with a .tor pseudo-TLD. The Internet DNS defines a hierarchy of administrative realms that are closely tied to the depth of each name. By contrast, OnioNS makes no such distinction; we let onion service administrators claim second-level names and then control all names of greater depth under that second-level name.

A **ticket** is a small and fundamental data structure. It contains *type*, *name*, *contact*, *rand*, *signature*, and *pubHSKey*. Tickets by default have *type* set to “ticket”, but this data structure becomes a **record** if *type* is set to any of the operations described in section 5.4.9.

A **mirror** is Tor router that is acting as a name server within the OnioNS network. Mirrors maintain a textual database of system information and respond to client queries but usually do not accept new DNS records or other information from onion services. We note that mirrors may be outside the Tor network, but this scenario is outside the scope of this work.

Quorum candidates are mirrors that provide proof in Tor's consensus documents that they hold a current copy of the database and that they have sufficient CPU and bandwidth capabilities to handle OnioNS communication in addition to their normal Tor duties.

The **Quorum** is authoritative subset of Quorum candidates who have active responsibility over the OnioNS database. Quorum nodes accept and process information from onion services but do not respond to client queries. The Quorum is randomly chosen from the set of Quorum candidates and is rotated periodically, as described in section 5.4.

L_Q	size of the Quorum
L_T	number of routers in the Tor network
Q_i	the ith Quorum where i is an iteration counter
Δq	lifetime of the Quorum
$r(f)$	if r is a record, the field f in r
$\|S\|$	the cardinality of the set S

Table 1. Frequently used notation.

5.3 Infrastructure

We embed OnioNS infrastructure within the Tor network by utilizing existing Tor nodes as hosts for OnioNS mirrors. Each Tor node may opt to run a onion services which then powers a OnioNS mirror running on localhost. As these onion services are part of OnioNS, they must be accessed by their traditional .onion address, but this is acceptable as these servers are never accessed directly by end-users. Our reliance on onion services allows us to recycle existing TLS links between Tor nodes and leverage Tor circuits to obscure all communication between end-users and OnioNS infrastructure without requiring a modification to the Tor executable. In essence, all communication with or within OnioNS is hidden from outside observers by ephemeral internal Tor circuits that need not pass through exit routers, increasing privacy and reducing our attack surface.

We authenticate servers in our infrastructure using Ed25519 [3] keys; as of Tor 0.2.7, Tor routers generate and manage Ed25519 keypairs and include their public key in the network consensus. We use Ed25519 because of its strength, size, and speed advantages over Tor's original RSA-1024 identity keys. OnioNS servers provide proof-of-knowledge of their private Ed25519 key for all outbound traffic on their onion service, achieving end-to-end authentication of all OnioNS communication. Throughout the remainder of this paper, "onion services" refers exclusively to onion services that are not part of the OnioNS infrastructure.

Each mirror maintains two distinct databases; "main" and "ephemeral", which both contain records. Newly received records are temporarily stored in the ephemeral database, which is periodically merged into the main database. Mirrors use their main database to respond to clients, who can then authenticate the responses against information published by Quorum nodes. Quorum nodes maintain an additional and secret database that contains lottery tickets.

5.4 Protocols

We now describe the protocols fundamental to OnioNS functionality.

5.4.1 Random Number Generation

We use $\mathcal{G}(t)$ as a basis for several of our protocols, although we note that $\mathcal{G}(t)$ has applications in Tor beyond OnioNS. One straightforward definition of $\mathcal{G}(t)$ is the SHA-384 hash of Tor's consensus documents. If the Tor network is dynamic enough to provide significant amounts of entropy into the consensus documents, then $\mathcal{G}(t)$ may be considered cryptographically secure. However, this assumption does not hold because current router descriptors are publicly available before the consensus documents are published, allowing $\mathcal{G}(t)$ under this approach to be easily manipulated by a few malicious Tor routers. The attack becomes significantly easier in the final moments before the directory authorities publish the consensus.

Instead, we suggest implementing $\mathcal{G}(t)$ as the commitment scheme proposed by Goulet and Kadianakis [10]. Their algorithm modifies the consensus voting protocol that is run once an hour by Tor directory authorities. In their scheme, at 00:00 UTC each authority commits a SHA-256 hash of a secret value v into each consensus vote across a 12 hour period. Then at 12:00 UTC, each directory authority reveals v across the next set of 12 consensuses. Then at 24:00 UTC, the revealed values are hashed together to create a single random number, which is then embedded in the consensus documents so that it is efficiently distributed to both Tor routers and clients. A different random number thus appears in the consensus every 24 hours. While this implementation of $\mathcal{G}(t)$ defines t as an integer of 24 hours, Δq may be greater than 24 hours, as discussed in 9.2. Therefore, throughout the remainder of this document we will use the notation $\mathcal{G}(i)$ to reference the $\mathcal{G}(t \cdot 24 \cdot \Delta q)$ that defines Q_i.

The two time boundaries in this implementation of $\mathcal{G}(t)$ trigger events within our system. The publication of $\mathcal{G}(i)$ at midnight defines the selection of the next Quorum, causes all mirrors to publish the state of their database, marks the beginning of the next lottery, and determines the winners of the previous lottery. The first reveal at 12:00 UTC ends the lottery and contains the pool of Quorum candidates. We clarify our distinction of these boundaries in section 6.1.

5.4.2 Authenticated Denial-of-Existence

We described in section **??** that a malicious name server may forge a response or may falsely claim non-existence of a name. These are attack vectors that remain open by naming systems that do not provide authentication mechanisms. We use a Merkle tree [14] to defend against these attacks with minimal networking costs. This tree is a fundamental authentication mechanism for both existing and non-existing names. All mirrors, including Quorum nodes, perform this algorithm. The tree's root hash is then checked by clients during other protocols.

1. Charlie fills an array list S with the $r_i(name) \| \mathcal{H}(r_i)$ for each record r_i received from onion services.
2. Charlie sorts S by the $name$ field.
3. Charlie constructs a Merkle tree T from S.
4. Charlie publishes the root hash of T in the consensus as described in section 5.4.3.

We note that a sorted Merkle tree does not support dynamic record updates and must be rebuilt at each update. While other data structures exist that support proof of existence and non-existence and allow efficient updates, such as a skip list [9], these structures are significantly more complicated. We consider it sufficient to use a Merkle tree as the tree is only rebuilt once per day in $\mathcal{O}(n \log(n)$ time.

5.4.3 Quorum Qualification

Quorum candidates must prove that they are both up-to-date mirrors and that they have sufficient capabilities to handle the increased communication and processing demands from OnioNS protocols, an additional burden on top of their traditional Tor responsibilities.

The naïve solution to demonstrating the first requirement is for all participants to simply ask mirrors for their internal database, and then compare the recency of its database against the databases from the other mirrors. However, this solution does not scale well; Tor has ≈ 2.1 million daily users [22]: it is infeasible for any single node to handle queries from all of them. Instead, at 00:00 UTC each day, let each mirror merge the ephemeral database into the main database, recompute the Merkle tree, and place the root hash inside the Contact field of its router descriptor so that the hash appears in the network consensus. The Contact field is typically used to hold the email address and PGP fingerprint of the router's administrator, but our use of the Contact field allows us to distribute the hash without modifying Tor infrastructure. Mirrors should also distribute their onion service address in the same way.

Tor provides a mechanism for demonstrating the latter requirement; Quorum candidates must have the Fast, Stable, and Running flags. Tor routers with higher CPU or bandwidth capabilities relative to their peers also receive a proportionally larger consensus weight from the directory authorities. This consensus weight in turn strongly influences router selection during circuit construction: routers with higher weights are more likely to be chosen in a circuit. This scheme also increases Tor's resistance to Sybil attacks. Thus, we can benefit from this infrastructure by selecting the Quorum from the pool of Quorum candidates by a similar mechanism.

5.4.4 Quorum Formation

Mirrors and Tor clients can check the aforementioned qualifications to locally derive the current or any previous Quorum in $\mathcal{O}(L_T)$ time locally without performing any network queries. Without loss of generality, let a client Alice run this algorithm. Alice must download consensus documents from some source, however these documents are timestamped and signed by Tor directory authorities and thus may be retroactively authenticated regardless of where they are archived.

1. Alice obtains and validates two consensus documents: C_a, which is published at 00:00 UTC and contains $\mathcal{G}(i)$; and C_b, which is the document published 12 hours prior to C_a.
2. Alice constructs a list S from C_b of Quorum candidates that have the Fast, Stable, and Running flags.
3. For each group $g \in S$ that publishes an identical root hash, Alice computes $s_g = \sum_{j=0}^{|g|} w_g(j)$ where $w_g(j)$ is the consensus weight of Tor router j in group g. The Quorum candidates, qc, is the group with the largest value of s_g.
4. Alice uses $\mathcal{R}(\mathcal{G}(i))$ to select $\min(\text{size}(qc), L_Q)$ Quorum nodes from qc with the probability of selecting router x determined by

$$P(x) = \frac{w_{qc}(x)}{s_{qc}}$$

5.4.5 Database Selection

The OnioNS network propagates information in near real-time in a peer-to-peer fashion; mirrors open authenticated circuits to other mirrors and subscribe for new tickets or records. All Quorum nodes subscribe to each other, forming a complete graph, and non-Quorum mirrors subscribe to all Quorum nodes. Under this scheme, all mirrors that remain online and at least semi-honest will process the same ephemeral and main databases. However; however, mirrors that drop offline will be out-of-date and must synchronize against the network by the following algorithm. Let Charlie be a mirror.

1. Charlie asks each Quorum node for $\mathcal{H}(db_e)$ and $\mathcal{H}(db_m)$, where db_e and db_m is the node's ephemeral and main databases, respectively.
2. Charlie finds the largest group, g, of Quorum nodes that return the same hashes.
3. Charlie downloads db_e and db_m from any node in g.
4. Charlie verifies the integrity of all records.
5. Charlie merges and db_e and db_m into his locally-stored db_e and db_m.

Quorum nodes that were temporarily offline conduct the same algorithm, but may also ask other Quorum nodes to replay new tickets so that they may update that database too.

5.4.6 Ticket Generation

An onion service administrator, Bob, may enter into the OnioNS lottery by generating a ticket, containing a second-level domain name for his onion service. The Quorum verifies the validity of his ticket and it may be further checked by mirrors and clients if his ticket wins the lottery, so Bob must follow this protocol to ensure that his ticket is accepted by all parties.

1. Bob sets *type* to "ticket".
2. Bob sets *name* to a meaningful domain name.
3. Bob sets *subdomains* to a map of domains of level three or higher and their respective destinations, which may be to either .tor or .onion domains.
4. Bob optionally sets *contact* to his PGP key fingerprint.
5. Bob sets *rand* to $\mathcal{G}(i)$.
6. Bob sets *signature* as the output of $S_{RSA}(\textit{type} \parallel \textit{name} \parallel \textit{subdomains} \parallel \textit{contact} \parallel \textit{rand}, r)$ where r is Bob's private RSA key.
7. Bob saves his RSA public key in *pubHSKey*.

Bob's ticket is valid when $\text{PoW}(\textit{signature}) \leq t_d$ where t_d is a difficulty threshold set by Quorum nodes. Each iteration of $\text{PoW}(\textit{signature})$ results in a different and one-way output because $S_{RSA}(m, r)$ is a probabilistic signature scheme. Bob must repeatedly resign and recompute $\text{PoW}(k)$ until the formula is satisfied. Once this is the case, Bob sends his ticket to all Quorum nodes.

5.4.7 Ticket Processing

A Quorum node $Q_{i,k}$ listens for tickets from onion service administrator. When a ticket t is received, $Q_{i,k}$

1. Rejects t if t is not valid according to the protocol described in section 5.4.6.
2. Rejects t if t's *name* already exists in its lottery, ephemeral, or main databases.
3. Rejects t if the onion service does not have a descriptor in Tor's distributed hash table.
4. Rejects t if the onion service cannot answer an HTTP GET request.
5. Otherwise, it accepts t, records t in its lottery database, and sends t to all other Quorum nodes.

5.4.8 Lottery Management

In section 5.4.6 we describe a proof-of-work (PoW) protocol that acts as a barrier-of-entry. It is straightforward to design a system where a name is awarded after the completion of this PoW; however such a system is vulnerable to attacks by adversaries with strong computational capabilities who may quickly register many names, as we described in section 4. In an attempt to resolve this problem, we introduce a lottery-like system, managed by Quorum nodes.

The lottery starts at the formation of the Quorum. During the lottery, the Quorum accepts requests (or "tickets") for names for onion services, creating a list T_i. The lottery ends when Q_i is cycled to Q_{i+1}. At this time, each node in Q_i performs the following algorithm. Let Charlie $\in Q_i$.

1. Charlie publishes T_i to all subscribers.
2. Charlie uses $\mathcal{R}(\mathcal{G}(i+1))$ to select a list of winners, W_i, from T_i.
3. Charlie merges W_i into his main database, letting them receive names.

All mirrors or other parties may verify W_i and update their main database accordingly through the same protocol because T_i is now public. As this algorithm occurs at 00:00 UTC, mirrors then update their Merkle root hash per the protocol described in section 5.4.3.

In order to defend against the one-to-many and many-to-one attacks described in section 4, each Quorum node only accepts the first ticket per onion service and the first ticket per name. We counter the many-to-many attack by setting the threshold t_d for ticket $i+1$ according to the formula $d_b / \lceil \frac{j}{|W_i|} \rceil$, where d_b is the base difficulty and j is the ticket's index, $1 \leq i \leq |W_i|$. Thus the Quorum accepts the first $|W_i|$ tickets at $t_d = d_b$, the second $|W_i|$ tickets at $t_d = \frac{d_b}{2}$, and so on.

5.4.9 Record Operations

OnioNS also supports common operations on names. Owners of named onion services may construct modify, renew, transfer, or delete records and issue the records to the Quorum. Once received, mirrors add these records to their ephemeral database. These records can be send to and authenticated by clients within 24 hours, as mirrors merge their ephemeral database into their main database and update their Merkle root publication at

00:00 UTC every day. In all cases, Bob sets the *type* field to the appropriate record type.

Bob can modify his registration by changing either his *subdomains* or *contact* fields. Bob may also transfer the registration to a new owner by issuing a transfer record, which contains *recipientKey*, the public RSA key of the new onion service. Bob may also relinquish control of his name by issuing a delete record. Bob does not need to recompute proof-of-work for any of these records as these operations are cheap for the Quorum to apply. However, OnioNS names expire after 30 days, so name owners must periodically renew registrations to maintain ownership. This can be done by issuing a renew ticket with an updated $\mathcal{G}(i)$ and recalculating the proof-of-work algorithm.

5.4.10 Domain Query

Alice only needs Bob's ticket or his latest record to contact Bob by his meaningful name. She then uses the Merkle tree structure to verify that her name server responds with the correct ticket or record, or to achieve authenticated denial-of-existence if her query has no corresponding data structure. Let Alice type a domain d into the Tor Browser.

1. Alice contacts a name server Charlie via his onion service.
2. Alice asks Charlie for a ticket or record r containing d.
3. Charlie extracts the second-level name n from d.
4. If r exists, Charlie returns r, the leaf node containing n, and all the nodes from the leaf to the root and their sibling nodes.
5. If r does not exist, Charlie returns two adjacent leaves a and b (and the nodes on their paths and siblings) such that $a(name) < n < b(name)$, or in the boundary cases that a is undefined and b is the left-most leaf or b is undefined and a is the right-most leaf.
6. Alice verifies the authenticity or non-existence of r by
   (a) Asserting that n is either contained in the subtree or that n is spanned by the subtree leaves, respectively.
   (b) Asserting the correctness of the hashes in the subtree.
   (c) Asserting that the root hash matches the hash published by the largest agreeing set of Quorum nodes.
7. If these assertions fail, Alice knows that Charlie is dishonest and she must repeat this protocol with a different mirror.
8. If d in r points to a domain d_2 which has a .tor pseudo-TLD, Alice jumps to 2 and queries for d_2.
9. Alice computes Bob's .onion address from $r(pubHSKey)$ and contacts him in the onion service protocol.

While Alice can verify the authenticity and uniqueness of r by synchronize against the OnioNS network and downloading the database from the Quorum, but this is impractical in most environments. Tor's median circuit speed is often less than 4 Mbit/s, [22] so for the sake of convenience data transfer must be minimized. Therefore Alice can simply fetch minimal information and rely on her existing trust of members of the Tor network.

5.4.11 Onion Query

OnioNS also supports reverse-hostname lookups. In an Onion Query, Alice issues an onion service address *addr* to Charlie and receives back all Records that have *addr* as either the owner or as a destination in their *subdomain*. Alice may obtain additional verification on the results by issuing Domain Queries on the source .tor domains. We do not anticipate Onion Queries to have significant practical value, but they complete the symmetry of lookups and allow OnioNS domain names to have Forward-Confirmed Reverse DNS matches. We suggest caching destination onion service addresses in a digital tree (trie) to accelerate this lookup; a trie turns the lookup from $\mathcal{O}(n)$ to $\mathcal{O}(1)$, while requiring $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space to pre-compute the cache.

6 Security Analysis

In this section, we analyse the security of the Onion Name System with regard to our security goals and expected threat model. We supplement with a statistical analysis of the Quorum in sections 9.1 and 9.2.

6.1 Global Randomness

We implement $\mathcal{G}(t)$ using [10]. Commitment protocols have been studied in other works [16, 23] and are well understood. If all parties are at least semi-honest then the commitment protocols generally display correctness, privacy, and binding. However, if some participants are malicious, they demonstrate known weaknesses. Namely, while reveals must demonstrably match commits, each participant may choose to reveal or not. If they do not reveal, their value is lost and the protocol produces a different output. If Mallory controls b participants, she can make this choice with each participant in turn, allowing 2^b different outcomes.

[10] relies on nine directory authorities, which are maintained by prominent members of the Tor community. The security of the Tor network rests on the assumption that five or more are at least semi-honest, thus their commitment scheme has at most $2^4 = 16$ different outcomes in the worst case without violation of our assumptions. Given the statistical calculations and the safety margin introduced by the recommendations in section 9.1, we do not consider this a significant threat to our system and conclude that the Quorum Formation protocol is secure under our design assumptions. The unpredictability of the reveals and the low probability of compromise shown in Figures 6 and 7 provides the strongest defense against Quorum-level attacks.

Assuming that all directory authorities reveal at 12:00 UTC, the design of [10] allows $\mathcal{G}(i)$ to be calculated up to 12 hours before $\mathcal{G}(i)$ is published in the consensus. If we select Q_i from the 00:00 UTC consensus containing $\mathcal{G}(i)$, an adversary could calculate $\mathcal{G}(i)$ to add or remove routers from the consensus in order to deterministically inject malicious routers into Q_i. As a countermeasure, the protocol in section 5.4.4 selects the Quorum from the consensus containing the first set of reveals.

6.2 Integrity Guarantees

Merkle trees are widely used to achieve secure verification of very large data structures. They are an integral component in the ZFS file system, Bitcoin, [15] Apache's Cassandra NoSQL database, [8] and in many other applications. The security of the Merkle tree largely rests on the underlying hash function and its resistance to second pre-image attacks. During a domain query, clients fetch a subtree from mirrors, verify the integrity of the ticket or record against the leaf node, and recompute and verify the hashes of the subtree. The cryptographic strength of SHA-384 prevents mirrors from forging or falsely claiming non-existence of a ticket or record. Clients also check the subtree value against the Quorum's published hashes, preventing mirrors from forging the subtree or returning an obsolete subtree. This approach provides strong integrity guarantees for both existent and non-existent records even if the mirror is malicious.

6.3 Proof-of-Work

We integrated the proof-of-work step into our registration protocols in order to create a barrier-of-entry, with the expectation that Bob run the $\mathrm{PoW}(k)$ function himself. However, our algorithm does not entirely prevent Bob from outsourcing this expensive computation to a secondary computational resource, Frank, whom we assume does not have the private key r.

First, Bob repeatedly computes $S_{RSA}(c, r)$, producing a large list of digital signatures. Frank tests $\mathrm{PoW}(x) \leq t_d$ for each digital signature x until the formula is satisfied. Bob then sets the ticket's *signature* field to this x value, producing a valid ticket. Bob can minimize the size of his precomputed list by sending each signature individually to Frank for testing. However, Frank must perform many calls to $\mathrm{PoW}(k)$, this still incurs a computational cost upon Frank that must be financially compensated by Bob. Thus, the algorithm always introduces a cost to Bob.

Alternative algorithms may more tightly couple the $\mathrm{PoW}(k)$ and RSA-PSS components and prevent Bob from outsourcing the proof-of-work step. The next generation of Tor onion services [21] replaces RSA-1024 keys with Ed25519, which generates deterministic signatures and utilizes SHA-512 as part of the signing process. As such, it may allow us to tightly combine proof-of-work and digital signatures. We will explore this in future work.

6.4 Lottery

Our lottery uses proof-of-work as a barrier-of-entry and awards names to a fixed number of onion services to limit abuse. However, adversaries with significant computational capabilities may still register many tickets. This may be done as part of a statistical denial-of-registration attack or an en-mass registration strategy. By contrast, we anticipate that honest onion services will only attempt to register a single name, therefore they may be assumed to be computationally weak. We consider this dichotomy when selecting the lottery's initial parameters.

Our lottery has a high degree of usability if it awards names a very large number of onion services. Although this increases its vulnerability to the aforementioned attacks, we introduce several straightforward countermeasures. First, we may preload OnioNS with a large set of "reserved" names by constructing a mapping between popular onion services and their self-declared name. Although these pre-existing onion services must still generate a lottery ticket, they win names immediately. This both increases the usability of our system and removes a significant incentive for land rush attacks. Second, we demonstrate that a large number of winners and the lottery's threshold-adjustment scheme provides a statistical defense against well-resourced adversaries.

We counter the potential for abuse by decreasing the proof-of-work threshold (thus increasing the cost) in a stepwise manner proportionally to the number of tickets submitted to the lottery. As described in section 5.4.8, Q_i sets the threshold, t_d, according to

$$t_d = \frac{d_b}{\lceil \frac{j}{|W_i|} \rceil} \tag{1}$$

where d_b is the base difficulty, W_i is the set of lottery winners, and j is the ticket's index in W_i. Although this makes it more difficult for Craig to generate many names, it does introduce an advantage for onion services that submit their tickets near the beginning of the lottery. This side-effect creates lower and upper bounds on the cost for Craig; he can generate many tickets at minimal cost if he submits them to Q_i before innocent users, but his cost is maximized if he submits them after innocent users.

Let the function $T(p)$ compute the maximum number of tickets that have a total cost p, where

$$T(p) = \frac{|W_i| \cdot \sqrt{1 + \frac{8 \cdot p}{d_b \cdot |W_i|}} - |W_i|}{2} \tag{2}$$

If the innocent users have a collective computational power P_I and Craig has power P_C, then Craig can compute C_t lottery tickets, where

$$C_t \in [T(P_I + P_C) - T(P_I), T(P_C)] \tag{3}$$

If P_I and P_C remain constant, Craig's upper-bound increases sub-linearly to $|W_i|$. The expected number of Craig's tickets winning the lottery, W_C, is given by

$$W_C = \frac{C_t \cdot |W_i|}{T(P_I + P_C)} \tag{4}$$

In the worst case, Craig submits his tickets first. In practice, we expect this to be unlikely because innocent users can submit tickets at the same time. Thus we anticipate that W_C will usually be between the upper and lower bounds. In all cases, higher P_I reduces W_C. However, if Craig is significantly powerful, we could expect $W_C > |W_i| - 1$ during a statistical denial-of-registration attack. The lottery's resistance to this attack is highly dependent on $|W_i|$. However, large $|W_i|$ also allows Craig to claim or squat many names. Based on the above equations, we suggest $|W_i| = 35$. We note that this value is adjustable and can be easily increased to meet a demonstrably higher demand from the Tor community.

We must also select a value for d_b. Low values for d_b provides a low barrier-to-entry, increasing usability, but it also increases the lottery's vulnerability to attacks from Craig. We consider that Craig that has access to efficient cloud computation, such as Amazon EC2. A compute-optimized instance, such as c4.large in California, costs appropriately 1 USD for 11 CPU core-hours on an Intel Xeon E5-2666 processor. In Table 2, we use cpubenchmark.net to convert this to the equivalent quantity of computation from a selection of Intel i7 CPUs, which are commonly found in high-end consumer-grade computers.

We anticipate that innocent onion services with high-end CPUs will take less than five or six hours to generate a lottery ticket. Even with lower-end CPUs, fewer utilized cores, or a higher difficulty, the proof-of-work algorithm will generate a valid ticket eventually. It may be possible to lower the difficulty if we can prevent innocent onion services from outsourcing computation by more tightly coupling proof-of-work to a digital signature algorithm. Finally, we suggest doubling the difficulty every two years so as to keep pace with a global average increase in computational power.

With a sufficiently high value of $|W_i|$ and a significant value for d_b, we believe that our lottery can resist a well-resourced adversary while simultaneously serving computationally-weak innocent parties.

6.5 DNS Leakage

Accidental leakage of .tor lookups over the Internet DNS via human mistakes or misconfigured software may compromise user privacy. This vulnerability is not limited to OnioNS and applies to any pseudo-TLD; Mohaisen and Thomas observed .onion lookups on root DNS servers at a frequency that corresponded to external global events and highlighting the human factor in those leakages [25]. Closing this leakage is difficult; arguably the simplest approach is to introduce whitelists or blacklists into common web browsers to prevent known pseudo-TLDs

CPU	Speed	vCPUs	CPU Hours	Wall Time
c4.large	24804	2	11	7.33
i7-5960X	16002	16	17.05	1.42
i7-5930K	13666	12	19.96	2.21
i7-4930K	13074	12	20.87	2.32
i7-3930K	12080	12	22.58	2.51
i7-4790K	11228	8	24.30	4.05
i7-6700K	11028	8	24.74	4.12
i7-4790	10034	8	27.19	4.53
i7-6700	9962	8	27.38	4.56

Table 2. The average time to complete the proof-of-work across various computational resources. These CPUs utilize hyper-threading; we assume that a single vCPU provides 75 percent of the power of a real core.

from being queried over the Internet DNS. Such changes are outside the scope of this work, but we highlight the potential for this leak.

7 Evaluation

7.1 Implementation

We have build a reference implementation of the Onion Name System in C++11 as a supplement to this work. We use JSON-RPC on top of microhttpd [20] (a small HTTP server library) for networking and the Botan [6] library for most cryptographic operations. We encode all the data structures in JSON; JSON is significantly more compact than XML, but retains user readability and its support of basic primitive types is highly applicable to our needs. Since JSON is usually not capable of carrying binary values, we encode all non-ASCII values (such as SHA-384 hashes) in base64. Our code is licensed under the Modified BSD License, identical to Tor, and is available for Linux through a software repository at our onion service, http://onions55e7yam27n.onion.

We divided our software into three parts: OnioNS-client, OnioNS-server, and OnioNS-HS, with OnioNS-common as a shared library dependency. OnioNS-client negotiates with the user's Tor software to MitM all requests for torified TCP streams. It filters for our .tor pseudo-TLD while allowing all other lookups (IPv4, .onion, or DNS) to pass unimpeded. Once it intercepts a request for a .tor domain, it performs a Domain Query and rewrites the domain to a .onion address before allowing Tor to bind the request to a circuit. This approach preserves backwards-compatibility, achieving a design objective and enhancing usability. The OnioNS-server is comparatively simpler; it binds to a localhost TCP port and performs our protocols. Tor router administrators must then configure their Tor to host a onion service and bind it to the same localhost port so that OnioNS-server may be reached from the outside. Finally, OnioNS-HS is a simple command-line utility that prompts the user for domain information, performs the proof-of-work, and uploads the ticket or record to the current Quorum. In all three cases, our software functions with minimal configuration.

7.2 Integration Test

We have deployed a small testing network of Quorum and mirror servers. We first created an onion service for our project, set up a small web server, and used Shallot to generate a semi-meaningful address, "onions55e7yam27n.onion". We then used OnioNS-HS to create a lottery ticket for "example.tor" and then to transmit it to the Quorum. This sole ticket won the lottery and we received our name. Our ticket then propagated through the network to mirrors. Finally, we installed our client software into Tor Browser 5.5, a fork of Firefox 38.4.0 ESR. We typed "example.tor" into the Tor Browser and the request was intercepted, resolved through a Domain Query, and rewritten to "onions55e7yam27n.onion". The Tor binary then communicated with our onion service and returned the contents back to the Tor Browser. This process did not require any further user input and occurred behind-the-scenes, so the Tor Browser retained the "example.tor" name in the address bar and in the mouse-over text for relative hyperlinks. We illustrate the result in Figure 3. The software performed asynchronously and allowed normal browsing to both the Internet and other onion services while it resolved "example.tor".

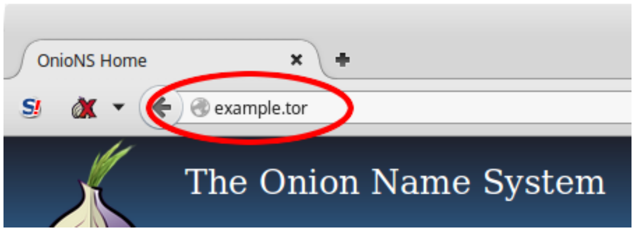

Fig. 3. We load our onionsite, available at "onions55e7yam27n.onion", transparently under the "example.tor" domain. The OnioNS software launches with the Tor Browser.

7.3 Results

7.3.1 Performance

We tested the performance of our OnioNS-HS and OnioNS-client software on two machines, M_a and M_b. M_a is powered by an Intel Core2 Quad Q9000 2.00 GHz Penryn CPU from late 2008 and M_b contains an Intel i7-6700K 4.0 GHz Skylake CPU from 2015, representing low-end and high-end consumer-grade computers, respectively. Both machines were hosted on a 16 Mbits connection, representing typical residential speed. We measured the performance of OnioNS-client. We averaged 200 samples of the CPU wall-time required for both machines to validate a record.

Description	M_a (ms)	M_b (ms)
Parsing JSON	5.21	2.42
$\mathcal{H}(x)$	4.35	2.15
$V_{RSA}(m, E)$	6.35	2.74
Total Time	15.91	7.31

The measurements show that clients can fully validate a record in less than 20 ms, even on low-end hardware.

7.3.2 Latency

Although Tor is a low-latency network, all communications occur over six-hop onion service paths through Tor, which introduces latency into most of our protocols. The exact round-trip latency is highly dependent on queuing delays, processing latency, and the speed of each router selected by either party, as well as the length and speed of the links between the routers. The latency is most significant for clients and adds an additional delay between the time that a user enters an OnioNS domain into the Tor Browser and the moment that Tor begins loading the onion service. Fortunately, latency and load times across Tor circuits have been well studied.

Domain Queries transfer a record/ticket and a Merkle subtree. We expect that a ticket or record would be less than 1 KiB. If OnioNS contains Z names, then the expected byte size of the serialized Merkle subtree, S_M, is approximated by

$$S_M = 2 \cdot (\log_{64} 2^{384} + e) \cdot \log_2 Z$$

where e represents markup and other formatting overhead in the transmission. Our implementation has $e \approx 98$ so if Z is 8192, then the expected networking cost for clients is $1664 + 1024 = 2688$ bytes, or 2.68 KB.

We conducted additional tests of latency and bandwidth. We conducted a total of 128 measurements from M_a to M_b across 16 unique Tor circuits. The client took between 1.4 and 4.2 seconds (averaging 3.4 seconds) to construct an ephemeral path to our onion service with an average 0.565 seconds of round-trip latency after the path was established. We observed an average bandwidth of 141 kB/s for large downloads. For small downloads, such as records and the Merkle subtree, the round-trip time will be dominated by the path latency. If the client builds the path on startup, we expect OnioNS lookups to add less than one second of latency in most cases. Our implementation provides a client-side cache, which helps to avoid redundant time-consuming queries to remote mirrors.

7.3.3 Usability

OnioNS is lightweight for clients, is straightforward for onion service administrators to use, and is easy for Tor router administrators to support. Our client software integrates well into the Tor Browser and starts and stops with the Tor Browser environment. As shown in Figure 3, OnioNS resolves and loads onion services under a meaningful name transparently without requiring any user interaction, similar to traditional DNS requests over Tor circuits. Domain Queries have low latency and the local cache can further reduce overhead, which is especially helpful if cached records are part of a chain of resolutions.

OnioNS-HS enables onion service administrators to claim a name and locally manage record information. The Ticket Generation protocol has minimal prompts: it asks the user for the main domain name, a list of subdomains and destinations, and an optional PGP key. It then loads the onion service private key, performs the proof-of-work algorithm, and automatically uploads the ticket to the Quorum. We released a beta test of our software to Tor developers and volunteers and received positive feedback on the simplicity of the registration process.

We achieve another primary usability benefit by introducing an automatic naming system for Tor onion services: it is no longer necessary for the Tor community to construct and maintain directories of onion services. The automatic resolution of domain names allows scaling beyond human-maintained directories. OnioNS should provide a significant usability enhancement to Tor users and the community at large.

7.4 Discussions

TODO: rewrite this However, we adapts some of the mechanisms in NSEC3, an extension of DNSSEC, to achieve authenticated denial-of-existence in this work. NSEC3 uses a sorted list of hashed names to prove non-existence; the client can quickly verify that no record exists between two names that canonically span the target. NXDOMAIN responses by DNS servers. Hashed Authenticated Denial of Existence **??** (NSEC3) is an extension for DNSSEC which authenticates NXDOMAIN responses by DNS servers. NSEC3 uses a sorted list of hashed names to prove non-existence; the client can quickly verify that no record exists between two names that canonically span the target. This mechanism also aims to avoid zone enumeration. In our work, we adapt this approach to also provide an authenticated denial-of-existence mechanism, but we tolerate enumeration.

In this section we further discuss and compare our work with related works. The Onion Name System and Namecoin both achieve all three properties of Zooko's Triangle and share some similarities, but the two systems have different threat models and distinct design objectives.

Namecoin's security rests on two primary assumptions: that its network is resistant to Sybil attacks and that more than 50 percent of the network's computational power is at least semi-honest. An attacker that controls the majority of the computational power (a "51% attack") or that has carried out a successful Sybil attack may then disrupt communications, corrupt the integrity of the blockchain, or can provide malicious responses to clients. Namecoin, like Bitcoin, also usually requires that clients download a complete copy of the blockchain. This introduces significant network, storage, and CPU costs which grow linearly to the age and popularity of the Namecoin network, preventing Namecoin from scaling to the general population. These costs can be largely avoided if the client uses a Namecoin-compatible DNS server or a Simplified Payment Verification [15] (SPV) scheme. These approaches are vulnerable to a variety of attacks if the name server is malicious and neither approach provides authenticated denial-of-existence.

By contrast, OnioNS' central security assumption is that Tor circuits provide privacy. This assumption implies that the majority of the directory authorities remain semi-honest and that Tor network remains resistant to Sybil attack and traffic analysis. We rely on these assumptions for all aspects of our system. Thus, we can use existing Tor infrastructure instead of introducing a new network, as Namecoin does. Moreover, if an attacker can compromise Tor circuits or de-anonymize onion services, then a privacy-enhanced naming system no longer becomes necessary or relevant. Unlike Namecoin, we do not rely on assumptions of computational power; while Craig can negatively impact the ability for innocent onion services to get a name, he cannot affect the security of OnioNS or disrupt network communications. Our system remains secure as long as the Quorum remains uncompromised and we demonstrate in the appendix that a sufficiently-large Quorum is strongly resistant to a Sybil attack on the Tor network.

OnioNS has significantly less overhead than Namecoin: our database can be modified or deleted in-place, mirrors only remember records for non-expired names, and the storage requirements for clients is constant. Instead, the client receives a Merkle subtree and checks the root against a single consensus document. The storage cost is therefore linear to the size of the Tor network and the networking cost is logarithmic to the number of registered names. The Merkle tree provides authenticated denial-of-existence and prevents a malicious server from acting dishonestly.

OnioNS and Namecoin both allow full enumeration of all registered domains; however we do not consider this a significant threat to our system as registrations do not contain personal information. Both systems operate under weaker adversarial models than GNS, which assumes than attacker may participant in any role, may infiltrate the network by large-scale Sybil attack, and is assumed to have more computational power than all honest participants combined. Neither Namecoin, OnioNS, nor Tor provide full defenses against such well-resourced adversaries. Tor onion services may become de-anonymized under GNS' adversarial model so we do not assume that our adversaries are that powerful.

8 Conclusions and Future Work

We have presented the Onion Name System (OnioNS), a decentralized, secure, and usable alternative DNS that maps globally-unique and meaningful names to onion service addresses, achieving all three properties of Zooko's Triangle. We enable any onion service administrators to anonymously claim a human-readable name for their server and clients to query the system in privacy-enhanced manner. We introduce mechanisms that let clients authenticate existent names and denial-of-existence claims with minimal overhead costs. Additionally, we utilize the existing and semi-trusted infrastructure of Tor, which significantly narrows our threat model to already well-understood attack surfaces and allows our system to be integrated into Tor with minimal effort. Our reference implementation demonstrates high usability and shows that OnioNS successfully addresses the major usability issue that has been with Tor onion services since their introduction in 2002.

Following publication, we will expand our implementation and pursuit deploying it onto the Tor network. OnioNS introduces new software and a new .tor pseudo-TLD but requires no changes to the Tor executable. OnioNS is also forwards-compatible to changes in Tor circuits or the onion service protocol and requires only small modifications to become compatible to [21], the proposed next generation of Tor onion services.

References

[1] Baruch Awerbuch and Christian Scheideler, *Group spreading: A protocol for provably secure distributed name service*, Automata, Languages and Programming, Springer, 2004, pp. 183–195.

[2] Daniel J Bernstein, *Dnscurve: Usable security for dns*, http://dnscurve.org/, 2009.

[3] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang, *High-speed high-security signatures*, Journal of Cryptographic Engineering **2** (2012), no. 2, 77–89.

[4] John Brooks, *Anonymous peer-to-peer instant messaging*, https://github.com/ricochet-im/ricochet, 2015.

[5] Ryan Castellucci, *Namecoin*, https://namecoin.info/, 2015.

[6] Botan Developers, *Botan: Crypto and tls for c++11*, http://botan.randombit.net/, 2015.

[7] Roger Dingledine, Nick Mathewson, and Paul Syverson, *Tor: The second-generation onion router*, Tech. report, DTIC Document, 2004.

[8] Apache Software Foundation, *The apache cassandra project*, https://cassandra.apache.org/, 2015.

[9] Michael T Goodrich, Roberto Tamassia, and Andrew Schwerin, *Implementation of an authenticated dictionary with skip lists and commutative hashing*, DARPA Information Survivability Conference &amp; Exposition II, 2001. DISCEX'01. Proceedings, vol. 2, IEEE, 2001, pp. 68–82.

[10] David Goulet and George Kadianakis, *Random number generation during tor voting*, https://gitweb.torproject.org/torspec.git/tree/proposals/250-commit-reveal-consensus.txt, 2015.

[11] George Kadianakis and Karsten Loesing, *Extrapolating network totals from hidden-service statistics*, Tech. report, The Tor Project, 2015.

[12] katmagic, *Shallot*, https://github.com/katmagic/Shallot, 2012.

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

[14] Ralph C Merkle, *A digital signature based on a conventional encryption function*, Advances in Cryptology-CRYPTO'87, Springer, 1988, pp. 369–378.

[15] Satoshi Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, Consulted **1** (2008), no. 2012, 28.

[16] Moni Naor, *Bit commitment using pseudo-randomness*, Advances in Cryptology—CRYPTO'89 Proceedings, Springer, 1990, pp. 128–136.

[17] Simon Nicolussi, *Human-readable names for tor hidden services*, Bachelor thesis, Leopold–Franzens–Universitat Innsbruck, Institute for Computer Science, 2011, http://www.sinic.name/docs/bachelor.pdf.

[18] Lasse Overlier and Paul Syverson, *Locating hidden servers*, Security and Privacy, 2006 IEEE Symposium on, IEEE, 2006, pp. 15–pp.

[19] Colin Percival and Simon Josefsson, *The scrypt password-based key derivation function*, Tech. report, September 2012, https://tools.ietf.org/html/draft-josefsson-scrypt-kdf-00.

[20] GNU Project, *Microhttpd*, https://www.gnu.org/software/libmicrohttpd/, 2015.

[21] The Tor Project, *Next-generation hidden services in tor*, https://gitweb.torproject.org/torspec.git/tree/proposals/224-rend-spec-ng.txt, 2015.

[22] ———, *Tor metrics*, https://metrics.torproject.org/, 2015.

[23] Ronald Rivest, *Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initializer*, Unpublished manuscript (1999).

[24] Nolen Scaife, Henry Carter, and Patrick Traynor, *OnionDNS: A seizure-resistant top-level domain*, IEEE Conference on Communications and Network Security (2015).

[25] Matthew Thomas and Aziz Mohaisen, *Measuring the leakage of onion at the root*, Tech. report, Verisign Labs, 2014.

[26] Matthias Wachs, Martin Schanzenbach, and Christian Grothoff, *A censorship-resistant, privacy-enhancing and fully decentralized name system*, Cryptology and Network Security, Springer, 2014, pp. 127–142.

9 Appendix

9.1 Quorum Size

In section 4, we assume that an attacker, Mallory, controls some fixed f_E fraction of routers on the Tor network. Quorum selection may be considered as an L_Q-sized random sample taken from an L_T-sized population without replacement, where the population contains $L_T \cdot f_E$ entities that we assume are compromised and colluding. Then the probability that Mallory controls L_E Quorum nodes is given by the hypergeometric distribution, whose probability mass function is shown in Equation 5. Mallory controls the Quorum if either $> \frac{L_Q - L_E}{2}$ honest Quorum nodes disagree or if $L_E > \frac{L_Q}{2}$. The former scenario is difficult to model theoretically or in simulation, but the probability of the latter may be calculated. If all Quorum nodes are selected with equal probability, then $\Pr(L_E > \frac{L_Q}{2})$ is given by the p-value of the hypergeometric test for over-representation, expressed in Equation 6.

$$\Pr(L_E) = \frac{\binom{L_T \cdot f_E}{L_E}\binom{L_T - L_T \cdot f_E}{L_Q - L_E}}{\binom{L_T}{L_Q}} \quad (5)$$

$$\Pr(L_E > \frac{L_Q}{2}) = \sum_{i=\lceil \frac{L_Q}{2} \rceil}^{L_Q} \frac{\binom{L_T \cdot f_E}{i}\binom{L_T - L_T \cdot f_E}{L_Q - i}}{\binom{L_T}{L_Q}} \quad (6)$$

Odd choices for L_Q prevents the network from splintering in the event that the Quorum is evenly split across two databases. We provide the statistical calculations of Equation 6 for various Quorum sizes in Figure 4.

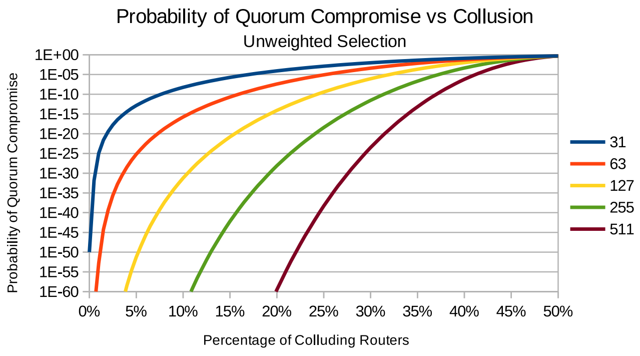

Fig. 4. The values for $\Pr(L_E > \frac{L_Q}{2})$ for Quorum sizes of 31, 63, 127, 255, and 511. All probabilities exceed 0.5 when more than 50 percent of the Tor network is under Mallory's control. We set our population to 4540 routers; the average number of routers with the Fast, Stable, and Running flags across all consensuses in July 2015 [22].

However, we select Quorum members according to consensus weight, akin to router selection in a Tor circuit. The distribution of consensus weight (and thus the selection probabilities) for routers with the Fast, Stable, and Running flags closely follows an exponential distribution, as shown in Figure 5. The figure suggests that the Tor network contains a low number of high-end routers and a large number of low-end routers.

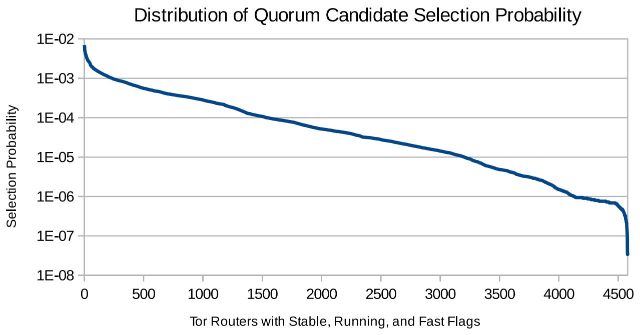

Fig. 5. The normalized distribution of consensus weights of Quorum candidates in July 2015, reflecting the probabilities of inclusion in the Quorum. The distribution may be modelled by an exponential trendline with $R^2 = 0.9884$, but appears slightly super-exponential.

We now re-examine Equation 6 with regard to this distribution of consensus weight. Consider that the hypergeometric distribution describes the probability of selecting k Mallory-controlled routers in an L_Q-sized Quorum from an N-sized population containing K Mallory-controlled routers. Let $L(x)$ be the probability distribution of selecting a router whose consensus weight is at the lowest x percentile. Then the probability of compromise is given by Equation 8 where K, the expected number of routers in a population of size N, is given by Equation 7, and R is the probability that routers outside $L(x)$ are compromised. Since $L(x)$ describes probabilities and N must be a natural number, $(N \in \mathbb{N})$ this approach provides an approximation of the probability of compromise.

We illustrate the probabilities against discrete values of x and various Quorum sizes in Figure 6 using $N = 4540$, consistent with the population in Figure 4.

$$K = N \cdot \left(\int_0^x (L(x)) + R \cdot \int_x^1 (L(x)) \right) \quad (7)$$

$$\Pr(L_E > \frac{L_Q}{2}) = \sum_{i=\lceil \frac{L_Q}{2} \rceil}^{L_Q} \frac{\binom{K}{i}\binom{N-K}{L_Q - i}}{\binom{N}{L_Q}} \quad (8)$$

In contrast to Figure 4 which demonstrates that an unweighted selection leads to a high probability of compromise with small levels of collusion, Figure 6 suggests that biasing Quorum selection by consensus weight provides a strong defence against large-scale Sybil attacks. Indeed, even when 60 percent of the low-end Quorum candidates are malicious, most Quorum sizes produce negligible probabilities of compromise. We consider it reasonable to assume that low-end routers are under

Mallory's control; these routers are the cheapest and logistically easiest to operate. Our approach remains resistant to this attack: these routers will be included in the Quorum very infrequently because of their low consensus weight.

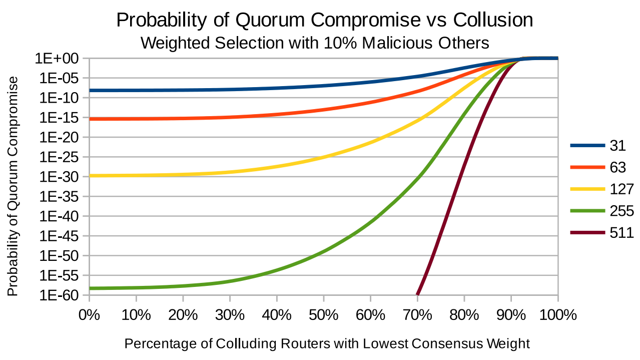

Fig. 6. The values for $\Pr(L_E > \frac{L_Q}{2})$ from Equation 8 for various Quorum sizes. We assume that all routers $\in L(x)$ are under Mallory's control, while routers $\notin L(x)$ have a 10 percent chance of being under Mallory's control.

Small Quorums are also more susceptible to node downtime or denial-of-service attacks. Figure 6 shows that the choices of $L_Q = 31$ is suboptimal; it is more easily compromised even with low levels of collusion. $L_Q = 63$ is more resistant, but not significantly more so. We therefore recommend $L_Q \geq 127$.

9.2 Quorum Rotation

In section 4, we assume that f_E is fixed and does not increase in response to the inclusion of OnioNS on the Tor network. If we also assume that L_T is fixed, then we can examine the impact of choices for Δq and calculate the probability of Mallory compromising any Quorum over a period of time t. Mallory's cumulative chances of compromising any Quorum is given by $1 - (1 - f_c)^{\frac{t}{\Delta q}}$ where f_c is Mallory's chances of compromising a single Quorum. We estimate this over 10 years in Figure 7.

Figure 7 suggests that although larger values of Δq positively impact security, the choice of L_Q is more significant. Furthermore, even "Stable" routers in the Tor network may be too unstable for very slow rotation rates, and small values for Δq also reduces the disruption timeline for a malicious Quorum. Therefore, based on Figure 7, we further reiterate our recommendation of $L_Q \geq 127$ and suggest $\Delta q = 7$. Although a malicious Quorum would have the capabilities to deploy a variety of attacks on the network, the proper selections of L_Q

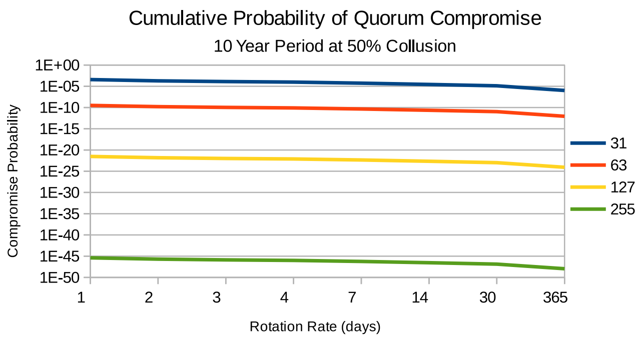

Fig. 7. The cumulative probability that Mallory controls any Quorum at different rotation rates over 10 years at $f_E = 50$ for Quorum sizes 31, 63, 127, and 255. We base these statistics on the probabilities from Figure 6 at 50 percent collusion.

and Δq reduces the likelihood of this occurring to near-zero probabilities. We consider this a stronger solution than introducing countermeasures to specific Quorum-level attacks.

9.3 Derivation of Lottery Analysis Equations

In this section we derive and further clarify the lottery analysis equations that we introduced in section 6.4. We set the proof-of-work threshold, t_d according to

$$t_d = \frac{d_b}{\lceil \frac{j}{|W_i|} \rceil}$$

Thus $t_d = d_b$ for the first $|W_i|$ tickets, $t_d = \frac{d_b}{2}$ for the second set of $|W_i|$ tickets, and so on. The sum of the coefficient is given by the nth triangular number.

$$t(n) = \binom{n+1}{2} = \frac{n \cdot (n+1)}{2}$$

We can then compute $T(p)$ by deriving Equation 2.

$$p = d_b \cdot |W_i| \cdot (\frac{n \cdot (n+1)}{2})$$

$$\frac{p}{d_b \cdot |W_i|} = \frac{n^2 + n}{2}$$

$$0 = n^2 + n - \frac{2 \cdot p}{d_b \cdot |W_i|}$$

$$n = \frac{-1 + \sqrt{1 - 4(\frac{-2 \cdot p}{d_b \cdot |W_i|} - 2)}}{2}$$

$$n = \frac{-1 + \sqrt{1 + \frac{8 \cdot p}{d_b \cdot |W_i|}}}{2}$$

$$T(p) = \frac{|W_i| \cdot \sqrt{1 + \frac{8 \cdot p}{d_b \cdot |W_i|}} - |W_i|}{2}$$