Jesse Victors, Ming Li, and Xinwen Fu

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 name 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.

DOI 10.1515/popets-2017-0003
Received 2016-05-31; revised 2016-09-01; accepted 2016-09-02.

1 Introduction

Tor [11] is a third-generation onion routing system and is the most popular low-latency anonymous communication network in use today. In Tor, clients 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 client'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. Fig. 1 illustrates how clients communicate with onion services.

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 services 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 55,000 onion

Jesse Victors: Cigital, Inc.
E-mail: kernelcorn@torproject.org
Ming Li: Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ
E-mail: lim@email.arizona.edu
Xinwen Fu: Department of Computer Science, University of Massachusetts Lowell, Lowell, MA
E-mail: xinwenfu@cs.uml.edu

BY-NC-ND

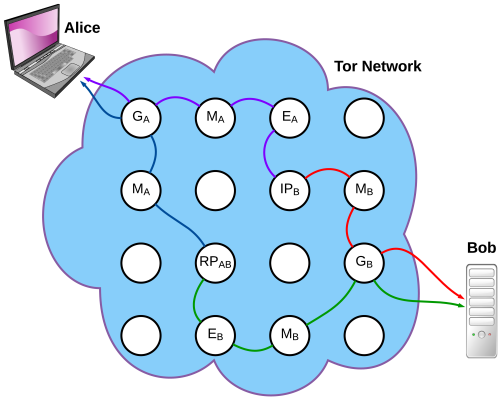

Fig. 1. A Tor client, Alice, and an onion service, Bob, first mate two Tor circuits (purple and red) at one of Bob's long-term *introduction points* (IP). They then renegotiate and communicate over another pair of Tor circuits (blue and green) at an ephemeral *rendezvous point* (RP). This achieves communication with bi-directional anonymity [21].

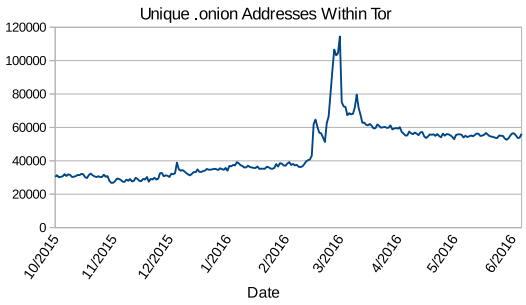

Fig. 2. The number of unique onion addresses seen in the Tor network between October 2015 and June 2016. [24].

services currently on the Tor network (Fig. 2) and the potential for continued growth both suggest the strong need for a more complete and wider solution to solve the usability issue.

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 a privacy-enhanced and verifiable manner. OnioNS is powered by a random subset of Quorum 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 proofs for authenticated denial-of-existence. We also design a novel lottery-based domain registration protocol to mitigate of denial-of-registration attacks and analyzed its security. 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.

Paper Organization: This paper is divided into eight main sections. In Section 2 we define our design objectives and explain why existing works do not meet our goals. In Section 3 we list prominent existing works. In Section 4, we 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 analyze the security of our assumptions and examine other attack vectors. In Section 7 we describe and demonstrate our implementation prototype and carry out performance analysis tests. Section 8 contains further comparisons with related works. We conclude in Section 9.

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 construct a distributed system and to providing a mechanism for authenticated denial-of-existence, which we describe in Section 5.5.2.

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 the authenticity of a domain-address pairing with cryp-

tographic guarantees. This objective provides a defense against phishing attacks from malicious name servers.

4. **Globally unique domain names**: Any domain name of global scope must point to at most one server. 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 [7] 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 [16]) 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, an 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 [20]. 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, 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 [28]. 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 [26] 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 does not require the user to install specialized software and it provides DNSSEC and other authentication mechanisms, the system is centralized by a single root server and thus vulnerable if the root is malicious or is compromised. Although there is a separation of duties in OnionDNS to allow for revocation should the main signing key be compromised, the revocation could take a long time. In contrast, our scheme is decentralized. In addition, in OnioNS we propose a lottery-based domain registration protocol, which effectively mitigates denial-of-registration attacks even when the attacker compromises and colludes with some Quorum nodes and has partial information about other registrants. Note that OnionDNS and OnioNS were named independently and readers should take care to not confuse the two works.

The GNU Name System [28] (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 con-

structed a decentralized peer-to-peer naming system [1], but like GNS, made no guarantee that domain names would be globally unique.

Namecoin [8] is an early fork of Bitcoin [19] 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. Users may also register DNS records or other information, which consumes Namecoins. 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 clients 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 so that consensus weight is an effective defense against Sybil attacks. We also assume that honest parties control the majority of Bitcoin's computational power, which implies that the Bitcoin blockchain is secure. We note that these assumptions are already made by Bitcoin and Tor and have held over time.

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 previous assumption. We assume a fixed percentage of dishonest and semi-honest routers; namely that the percentage of routers under 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 attack 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 and attempt to register many names.
2. Create one onion service and attempt to register many names.
3. Create many onion services and attempt to register a single name.
4. Create one onion service and attempt 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 impossible to construct a system that differentiates and selects between a single actor performing the first scenario and many actors performing the fourth scenario. However, we expect innocent actors to follow the fourth use case such that one entity (considering load-balancing) hosts one onion service.

5 Design of OnioNS

5.1 Cryptographic Primitives

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

- $\mathcal{H}(x)$ is a cryptographic hash function of a message x. We define $\mathcal{H}(x)$ as SHA-256.

- $S_{RSA}(m, r)$ is 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.
- $V_{RSA}(m, S, R)$ validates an RSA digital signature by accepting a message m, a signature S, and a public key R, and return true if and only if the signature is valid.
- PoW(x) is a one-way function that accepts an input key k and returns a deterministic output. While PoW(x) could ideally be set to memory-hard key derivation function such as scrypt [22], for performance reasons we define PoW(x) as $\mathcal{H}(x)$.
- $\mathcal{G}(t)$ is a cryptographically-secure beacon of random numbers. $\mathcal{G}(t)$ periodically returns a random number at time t, which is unpredictable before t but is publicly verifiable after t.
- $\mathcal{R}(s)$ is a cryptographically-secure pseudorandom number generator (CSPRNG) that accepts an initial seed s and returns a list of pseudorandom numbers. In our design, $s = \mathcal{G}(t)$. For efficiency reasons, we suggest AES in CTR mode using s as a key with a fixed IV.

5.2 Definitions

This section lists commonly-used terms and Table 1 defines frequent mathematical notation.

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*, *secondaryAddrs*, *subdomains*, *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 5.5.7.

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 hold a current copy of the database. They also have sufficient CPU and bandwidth capabilities to handle OnioNS communication in addition to their normal Tor duties.

The **Quorum** is an authoritative subset of Quorum candidates who have active responsibility to maintain 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 5.5.

$\lvert S\rvert$	the cardinality of the set S
$\lvert T\rvert$	number of routers in the Tor network
$\lvert Q\rvert$	size of the Quorum
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

Table 1. Frequently used notations.

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 an onion service which then powers an OnioNS mirror server 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 reuse 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, increasing privacy and reducing our attack surface.

We authenticate servers in our infrastructure using Ed25519 [3] keys. Starting with 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 use their private key to digitally sign outbound traffic from their onion service, achieving end-to-end authentication of all OnioNS communication.

5.4 Overview

The data flow in OnioNS is illustrated in Fig. 3. First, all parties retrieve $\mathcal{G}(t)$ (a beacon) from the Bit-

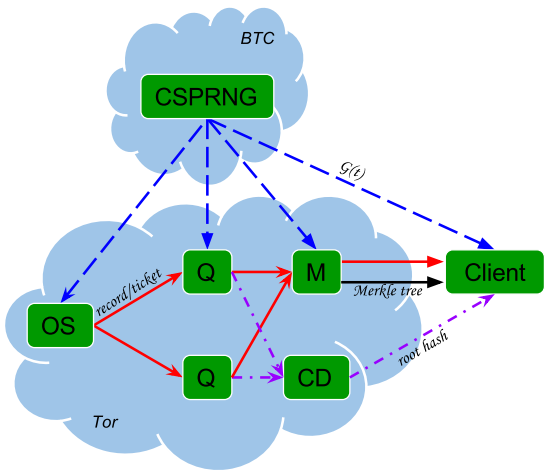

Fig. 3. An overview of data flow in OnioNS.

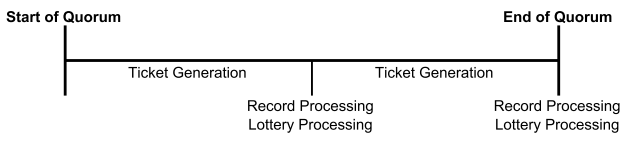

Fig. 4. The series of events within each Quorum's lifetime.

coin network, shown via a blue dash line. They then use CSPRNG $\mathcal{R}(\mathcal{G}(t))$ to select a set of Quorum nodes (Q) within the Tor network. Second, an onion service (OS) sends tickets or records to the Quorum. Since mirrors (M) subscribe to Quorum nodes, this data also propagates to mirror nodes. Each mirror maintains a Merkle tree of all the records obtained from the Quorum. At a later point, a client can query the mirror for the record. The mirror returns the ticket or record along with a path through the Merkle tree, shown in solid black. The client then verifies the Merkle tree proof and checks that the majority of Quorum published the same root in the network consensus document (CD), shown in the purple dash-dotted line. This procedure occurs much the same way even if no such ticket or record exists for a given name.

Next, we explain the series of events within each Quorum's lifetime. A set of Quorum nodes' lifetime can span one or multiple registration periods, each being 24 hours long, wherein onion services apply for a meaningful name by generating a ticket and performing proof-of-work. At the end of each registration period, Quorum nodes process the tickets received during that period, and use a new value of $\mathcal{G}(t)$ to determine a subset of registrants that have won the lottery. These onion services then receive their meaningful name and their records are added to the Quorum's and mirrors' name databases. This cyclic process is illustrated in Fig. 4.

5.5 Protocols

We now describe the protocols fundamental to OnioNS functionality. These protocols are listed according to their approximate order of execution in OnioNS.

5.5.1 Random Number Generation

$\mathcal{G}(t)$ is used as a basis for several of our protocols. The key issue is how to obtain a secure and decentralized source of randomness. One straightforward definition of $\mathcal{G}(t)$ is the SHA-256 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.

Another alternative approach is the commitment scheme proposed by Goulet and Kadianakis [15]. 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. Finally, 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.

However, the above commitment scheme has a well-known weakness when some of the directory authorities are malicious. 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, which can be used for Mallory's favor, e.g. skew the RNG output to select malicious and colluding Quorum nodes. Since preventing such attack requires trust in all nine directory authorities, which is contradictory to our goal of decentralization, we will not use this approach.

Instead, we propose to construct $\mathcal{G}(t)$ using the latest block in the longest (and most secure) Bitcoin blockchain. Bitcoin blocks consist of an 80-byte header and an array of transactions. The block header con-

tains a version number, timestamp, current difficulty level d, a 32-bit nonce, a SHA-256 hash of the previous block header, and a SHA-256 root hash of the Merkle tree constructed from all transactions in that block. Among these items, the nonce and Merkle hash are the main contributors of entropy for the block header, since the rest are predictable given all the previous blocks. According to the recent analysis by Bonneau, Clark, and Goldfeder [6], the amount of (computational) min-entropy contained by a block header is d bits, (currently $d > 68$) from which one can extract a near-uniform random number of at least 32 bits long. Since the blockchain's security relies on the assumption that honest parties control the majority amount of computation power, the head of blockchain can be regarded as a decentralized random source. We present more detailed analysis of security on this topic in Sec. 6.

Following the suggestion by [6], we construct a random beacon from the head of the blockchain as follows:

$$\mathcal{G}(t) = \mathsf{Ext}_k(B_t || \mathcal{H}(B_t)), \tag{1}$$

where B_t is the header of the latest mined block (at time t) from the longest chain in Bitcoin, and Ext_k is a standard randomness extractor such as HMAC, with k as a key to randomly choose the extractor function. In Bitcoin, such a valid block is mined about every 10 minutes by design.

Tor clients and Tor routers can acquire or calculate $\mathcal{G}(t)$ in several ways. First, they can download the entire Bitcoin blockchain from the Bitcoin network, find the block at time t, and generate $\mathcal{G}(t)$ from its header. However, this approach trades high security for significant storage, bandwidth, and processing costs, which may be impractical or prohibitive in many cases. The second method is for clients to query a trusted source for an initial "seed block" on the main blockchain and then download the remainder of the blockchain from that block. They can either download the blockchain peer-to-peer, from the Bitcoin network, or from a centralized server, but this choice does not matter since the chain is verified against the seed block. Periodically, clients could then reset the seed block to the latest block and repeat the process. This offers a significant reduction in overhead, but requires a trusted source for the initial seed block. The third, least secure, but cheapest method is to download the header from a trusted source. We allow Tor clients and Tor routers to perform any of these methods. We provide an onion service server for the seed block and header and note that we could also distribute $\mathcal{G}(t)$ through the consensus documents.

5.5.2 Authenticated Denial-of-Existence

We described earlier 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 [18] 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. Let Charlie be a mirror.

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 5.5.3.

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 [14], 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.

5.5.3 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 approximately 2.1 million daily users [24]: it is infeasible for any single node to handle queries from all of them. Instead, at 00:00 UTC each day, let each mirror apply any record operations that it received in the last 24 hours, 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 can

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.

5.5.4 Quorum Formation

Once OnioNS mirrors and Tor clients have $\mathcal{G}(t)$, they can check the aforementioned qualifications to locally derive the current or any previous Quorum in $\mathcal{O}(|T|)$ time locally without performing any additional network queries. Without loss of generality, let a client Alice run this algorithm at 00:00 UTC.

1. Alice obtains and validates the consensus document C published at 00:00 UTC. Since consensus documents are timestamped and signed by Tor directory authorities, Alice may download the document from any source without loss of security.
2. Alice obtains $\mathcal{G}(t_1)$ where $t_1>$ 00:00 UTC, i.e. the first beacon obtained after midnight, using the method from Section 5.5.1.
3. Alice constructs a list S from C of Quorum candidates that have the Fast, Stable, and Running flags.
4. 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.
5. Alice uses $\mathcal{R}(\mathcal{G}(t_1))$ to select $\min(\text{size}(qc), |Q|)$ Quorum nodes from qc with the probability of selecting router x determined by

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

For security purposes, Alice must apply $\mathcal{G}(t_1)$ to a consensus document published at time t_2, where $t_1 > t_2$. If $t_1 < t_2$, then an attacker who controls x Quorum candidates can maliciously influence Quorum selection after seeing the beacon by adding or removing some candidates from the consensus. We avoid this attack by retroactively applying the CSPRNG to an older consensus document.

5.5.5 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 and 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 information. 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 the SHA-256 hash of all records that the node has received.
2. Charlie finds the largest group, g, of Quorum nodes that return the same hashes.
3. Charlie uses delta compression to download recent records from any node in g.
4. Charlie verifies the integrity of all records and is fully synchronized if all records pass inspection.

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 process the lottery as part of the Domain Registration protocol.

5.5.6 Domain Registration

To prevent malicious domain registrations such as land-rushing and denial-of-registration attacks, we need to enforce some cost when an onion service administrator registers a domain name. A common way to do so in a distributed system is through proof-of-work (PoW) [19]. A PoW algorithm is usually a cryptographic challenge which is difficult to solve but easy to verify. In OnionDNS [26], the authors proposed an auction-based PoW mechanism in which the registrant who spent the most computing time for PoW wins a domain. It forces the registrants to focus their computing power on a small subset of domains rather than many at a time. However, it assumes that the game has incomplete information, i.e. participants do not know who their opponents are in each game when they are bidding on a domain. If the game participant information is fully known to the registrants, then for games with no honest registrants, an attacker can always solve the puzzle at the minimum difficulty and therefore successfully register many names. This makes it inapplicable to our

system because some of the Quorum nodes may be compromised and collude with powerful attackers trying to register many names.

To resolve the above challenge, we propose a lottery-based scheme. The key idea is two-fold: (1) require every registrant to submit a "ticket" and solve a PoW problem related to this ticket, where we enforce a threshold difficulty level of PoW for all the registrants as a barrier-of-entry; and (2) employ a weighted lottery drawing on the valid tickets to determine the final winners, so as to limit the rate of domain registration, where the weight is proportional to the computational effort spent by each registrant. The scheme consists of two phases: first, ticket generation and submission, and second, a lottery.

Ticket Generation and Submission. The ticket generation period spans for a 24-hour period: from 00:00 UTC each day until 24:00 UTC $-\delta t$, where δt is a small time period (e.g., five minutes). Starting from 00:00 UTC, an onion service administrator (a.k.a. registrant), Bob, may enter into the OnioNS lottery by generating a ticket, containing a second-level domain name for his onion service. Then, all the registrants submit the hash commitment of their tickets during the submission time window [00:00 UTC, 24:00 UTC $-\delta t$], and reveal their tickets after 24:00 UTC.

Bob generates an initial ticket by defining the following fields:

- *type*: "ticket".
- *name*: a meaningful domain name.
- *secondaryAddrs*: a list of additional .onion destinations for load-balancing across multiple servers
- *subdomains*, a map of domains of level three or higher and their respective destinations, which may be to either .tor or .onion domains.
- *contact*: (optional) Bob's PGP key fingerprint.
- *rand*: $\mathcal{G}(i)$.
- *nonce*: a nonce, which is a solution to the PoW problem.
- *pubHSKey*: Bob's RSA public key.
- *signature*: output of $S_{RSA}(type \parallel name \parallel secondaryAddrs \parallel subdomains \parallel contact \parallel rand \parallel nonce, r)$ where r is Bob's private RSA key.

In the above, *nonce* should be a solution to the following PoW:

$$v_{PoW} = \mathcal{H}(signature||nonce) \leq v_{th}, \tag{2}$$

where $v_{th} \in \mathbb{Z}$ and $v_{th} < 2^{256}$. We use $d_{th} = \lfloor log_2(2^{256}/v_{th}) \rfloor - 1$ and $d_{bob} = \lfloor log_2(2^{256}/v_{PoW}) \rfloor - 1$ to denote the threshold and actual difficulty levels, respectively (i.e., number of leading zero bits in the hash).

Note that Bob's tickets are valid only when d_{Bob} is larger than a difficulty threshold set by Quorum nodes. Each iteration of the above PoW results in a different and one-way output because $S_{RSA}(m, r)$ is a probabilistic signature scheme. Bob must repeatedly resign and recompute PoW until the formula is satisfied. Note that this discourages Bob to outsource the PoW computation to a powerful cloud service, since computing the signature requires possession of Bob's private key. Once the threshold difficulty is met, Bob must continue to search for an answer to the PoW such that it reflects the maximum difficulty (minimum v_{PoW}) he can obtain during the ticket generation period. No tickets are submitted during the ticket generation period in order to prevent malicious registrants from knowing the ticket information of other registrants in advance. During the submission time window (e.g., the last minute of the day), Bob finalizes the ticket corresponding to the solution to the PoW with minimum v_{PoW} he obtained so far and generates a hash commitment of everything inside his ticket: $commit_{bob} = \mathcal{H}(type \parallel name \parallel secondaryAddrs \parallel subdomains \parallel contact \parallel rand \parallel nonce \parallel pubHSKey \parallel signature)$. Then he sends $commit_{bob}$ to all the Quorum nodes. No tickets that are submitted after the ticket generation period will be accepted for the current day's lottery. After all the tickets are submitted, all the registrants reveal their ticket contents to the Quorum. These reveals form an initial pool of tickets.

Ticket Processing. Following the ticket reveals, each Quorum node $Q_{i,k}$ inspects every ticket in the initial pool, finds the actual difficulty for the PoW in each ticket by recomputing $v_{PoW} = \mathcal{H}(signature||nonce)$, and performs the following to prune the tickets to form a lottery ticket pool:

1. Rejects t if the hash of the record fields do not match any commitment hash.
2. Rejects t if the t's PoW value is greater than the threshold, i.e. $v_{PoW} > v_{th}$.
3. Rejects t if the record's signature is invalid or if any other field is malformatted.
4. Rejects t if t's *name* is already registered.
5. Rejects t if the onion service does not have a descriptor in Tor's distributed hash table.
6. Otherwise, it records t in its lottery pool, T_i.

Lottery Management. The lottery phase starts after ticket processing. Note that name collisions may exist in the lottery pool: multiple tickets claiming for

the same name. We resolve these collisions and ensure one-to-one correspondence between names and tickets in this phase. Each Quorum node performs the lottery phase by 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 winning tickets, W_i, from T_i. The detailed process is as follows. For each ticket $t_j \in T_i$, associate a weight inversely related to its actual PoW value obtained ($w_j = \lfloor 2^{256}/v_j \rfloor$), where $v_j = \mathcal{H}(signature||nonce)$). Compute a probability as $p_j = \frac{w_j}{\sum_{j=1}^{n} w_j}$. The lottery is drawn (without replacement) among all the tickets with their corresponding probabilities, with the rule that once one ticket for a name is chosen, all the other tickets for the same name are removed from the lottery pool. This goes on until $|W_i|$ winners are drawn.
3. Charlie allows all members of W_i to receive their names.

Although tickets are initially blinded, the above algorithm is publicly verifiable. All mirrors, clients, or other parties may verify W_i since T_i is public. As this algorithm occurs at 00:00/24:00 UTC, mirrors then update their Merkle root hash per the protocol described in Section 5.5.3.

In order to prevent land-rushing attacks we adopt a similar bootstrapping method as in OnionDNS. We 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. Similarly, we can also reserve popular Internet domain names. This approach both increases the usability of our system and removes a significant incentive for land rush attacks.

The lottery weight for each ticket is essentially proportional to the amount of computation power that its creator spends on the PoW, since the more CPU cycles it spends, the less the PoW value v it can obtain. Thus, if anyone wishes to win a single name's ticket, she needs to focus all its computation power to solve its PoW. This is the strategy that the legitimate parties will adopt. This algorithm is also resistant to denial-of-registration attacks: if an attacker tries to prevent legitimate onion services from registering a set of names, the attacker must spend as much computation on those tickets' PoWs to ensure they have a statistical advantage to win those names. If they do not specifically target any names and just randomly register many names at once, they need to spread their computation power over many names and each of those tickets will have low weight. Our analysis shows that such attack's impact is bounded in terms of how many names they can win, which is ensured by the rate-limiting lottery design and the difficulty threshold.

Another desirable property of the above protocol is that it prevents an attacker that colludes with a compromised Quorum node from obtaining information about the other registrants before the ticket submission phase, including the number of them, their claimed names, and PoW difficulty/answers. During the ticket submission, attackers may learn the total number of registrants, however others' claimed names and PoW answers are still hidden by the commitment scheme. Also, since ticket submission window is short (e.g., 5 minutes), it will not be enough for the attacker to make any meaningful changes to his own ticket (e.g., to compute another PoW answer with significantly higher difficulty). Therefore, the ticket generation and submission phase ensures a uniform information game for all the registrants.

5.5.7 Record Operations

OnioNS also supports common operations on names. Bob, an owner of an onion service, may construct modify, renew, transfer, or delete records and issue the records to the Quorum. In all cases, Bob sets the *type* field to the appropriate record type. Once received, mirrors hold the record in a queue and apply them at 00:00 UTC each day by the protocol specified in Section 5.5.3. Thus updates take up to 24 hours to propegate through OnioNS.

Bob can modify his registration by changing *secondaryAddrs*, *subdomains* or *contact* fields. Bob may also transfer the registration to a new owner by issuing a transfer record, which contains an additional field: *recipientKey*, the public RSA key of the new onion service. This transfer request can be authenticated since Bob's record is signed, similar to a Bitcoin transaction. Bob may also relinquish control of his name by issuing a deletion 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 90 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. The PoW requirement here reduces the risk of name squatting.

5.5.8 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 root hash published by the largest agreeing set of Quorum nodes, by validating every Quorum node's signature of its published root hash.
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 $r(pubHSKey)$ or randomly selects an .onion address from $r(secondaryAddrs)$ (if present), proceeding in a round-robin fashion until she contacts Bob over the onion service protocol.

It is impractical for Alice to download the whole database from all Quorum nodes over slow 6-hop onion service circuits. Alice uses the Merkle tree to verify the authenticity (or non-existence) and uniqueness of r with minimal networking costs even when the mirror Charlie is dishonest. This in turn requires trusting that the largest agreeing subset of the Quorum has published the correct root hash. We show in our security analysis that this assumption holds in most cases even if the Quorum is partially compromised.

Note that, although Alice needs to check all the Quorum nodes' signatures of the root hash, she does not need to do so for every Domain Query. Instead, this can be done once every registration period (24 hours) after which the records are updated. In this way, the signature verification cost is amortized.

5.5.9 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 analyze the security of the Onion Name System with regard to our security goals and threat model. We supplement with a statistical analysis of the Quorum in Appendix A.1 and A.2.

6.1 Global Randomness

The beacon $\mathcal{G}(t)$ should satisfy several security properties: unpredictable, unbiased, universally sampleable, and publicly verifiable [6]. The latter two properties follow directly from the construction. The first two depends on the security of the Bitcoin blockchain. The unpredictability can be quantified by the (computational) min-entropy of the beacon, while the unbiased property is defined by the statistical closeness of $\mathcal{G}(t)$ to an m-bit uniformly random string. In [6], the authors showed that this beacon generates 32 near-uniform random bits every 10 minutes. This would be sufficient for our system: if we assume $|Q| = 127$, then each Tor router is sampled at least $\approx 2\%$ chance to be in a Quorum, which is much higher than 2^{-32}.

The above assumes normal operation when honest parties control the majority of the computational power

in the Bitcoin network. Next, we analyze the security of this assumption itself. Bitcoin miners, who perform the PoW, are rewarded with an agreed-upon bounty for generating a new block as well as the cumulation of transaction fees. These provide an incentive to solve the proof-of-work problem. While miners originally attempted to solve the PoW problem independently, a community of miners may organize themselves into a "mining pool". Each miner in the pool attempts to find a solution to the PoW problem in the common block. When the PoW is solved, the reward is split across miners according to contributed processing power [12]. Since this is more financially rewarding, the vast majority of Bitcoin's hashrate is performed in this way. The top seven mining pools are shown in Fig. 2.

F2Pool	**27%**
AntPool	**19%**
BTCC Pool	**15%**
BW.COM	**10%**
BitFury	**10%**
Kano CKPool	**4%**
Slush	**4%**
(All others)	**12%**

Table 2. Distribution of the top seven mining pools in the Bitcoin network as of June 2016 [5].

As of June 2016, the aggregrate Bitcoin hashrate is 1.446 EH/s, or 1.446×10^{18} hashes/second [4]. An attacker could compromise the security of the Bitcoin blockchain through a 51% attack, wherein the attacker controls the majority of the hashrate and can thus control the longest chain. This can occur if either F2Pool, AntPool, and BTCC Pool collude (as they would then control 61% of the hashrate) or if an attacker introduce more than 1.446 EH/s of computational power into the network. The most cost-effective and energy-efficient ASIC miners are capable of up to 4.7 TH/s, 0.25 watts/GH, and an initial up-front cost of $520 USD [17]. Based on these figures, we estimate that a 51% attack would cost $158 million USD and would draw 395 megawatts. Therefore, so long as F2Pool, AntPool, and BTCC Pool do not collude, the honest majority computation power assumption in Bitcoin holds.

We also must analyze whether this beacon is resistant to manipulation attacks. In [6], the authors considered the scenerios where the attacker bribes the miners in order to suppress certain valid blocks and thus make the beacon predictable or otherwise favorable to the attacker. They model a manipulation-resistant lottery scheme as a finite-state Markov process and found that the attacker must possess high stakes in the outcome of a lottery to be economically feasible. In this paper, we apply the same methodology to analyze the security of the beacon. We can model the Quorum generation as a single-stage lottery where $|Q|$ nodes are chosen at a time using one beacon (i.e., $|Q|$ numbers derived from $\mathcal{G}(t)$). The attacker controls the Quorum only if Q contains the majority of malicious nodes. Thus, it has a binary reward function:

$$A(s) = \begin{cases} W & : \quad s \in S_* \\ 0 & : \quad s \notin S_* \end{cases} \tag{3}$$

where W is the reward of controlling the Quorum and S_* is the state set of all Quorum node combinations where malicious nodes gain the majority. Denote p the probability that attacker wins an unmanipulated lottery (the Quorum compromise probability, or the aggregated probability of states S_*), then p is very small according to our analysis in Appendix A.1. For instance, when $|Q| = 63$, percentage of colluding routers is 20%, $p = 5.5 \times 10^{-16}$ (referring to Fig. 8). Thus, following Eq. (4) in [6], the attack is advantageous whenever $W > \frac{1}{p}$ Bitcoins. This amounts to $\$4.3 \times 10^{17}$; suppose there are $50,000$ onion names in Tor as of now, each of them needs to worth $8.6 trillion. This means it is practically infeasible for the attacker to bribe the miners to manipulate the beacon and Quorum generation in OnioNS.

6.2 Integrity Guarantees

Merkle trees are widely used to achieve secure verification of very large data structures. The security of the Merkle tree 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 second pre-image attack resistance of SHA-256 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.

In the unlikely scenario that the attacker controls the majority of nodes in a Quorum, the attacker can modify or substitute existing records during that Quorum's lifetime, which can mislead clients. However, a malicious Quorum cannot overwrite history (e.g. records from past Quorums) because the previous Quorums maintain a read-only copy of its own historical records. These historical records can also be stored by other mir-

rors, the current Quorum, or an external server. A client can send the same query to past Quorums to see if the records differ. If they do and the current Quorum does not provide explanatory proof (such as a modification or transfer record) then the client can detect expired/recaptured domains or malicious substitutions and should warn the user. We introduce a flag into our software for this purpose, which will also serve as a deterrent to malicious Quorums.

Tor's resistance to Sybil attack helps to ensure the integrity of the Quorum and the security of OnioNS in general. New routers must demonstate long uptime and high reliability before they receive notable consensus weight. The arguably most prominent Sybil attack occured in December 2014 when members of the Lizard Squad group introduced approximately 3,300 new Tor routers. Although Lizard Squad then controlled more than half of the size of the Tor network, their routers collectively had 0.2743% of the consensus weight. The Sybil attack was noticed by the Tor community and the offending routers were banned from the network. Small-scale Sybil attacks have also been detected through community-developed tools such as SybilHunter [30].

6.3 Lottery

Our lottery uses proof-of-work as a barrier-of-entry and awards names to a fixed number of onion services in any period to limit abuse. Here, we consider powerful adversaries with significant computational capabilities who may register many tickets as part of a land rush or to perform a denial-of-registration (DoR) attack. On the other hand, we anticipate that honest onion services will only attempt to register a single name, therefore they may be assumed to be computationally weak. We analyze our scheme's resilience against attacks in Section A.3. We found that with appropriate selection of the difficulty threshold, the attacker's expected number of winning names is approximately equal to the total number of winners scaled by the portion of its computation effort over the combined effort of attacker and all the legitimate registrants. Here, we recommend an initial value for the difficulty threshold, d_{th}.

Our analysis suggests that to effectively mitigate DoR attacks, we should set the difficulty level of innocent onion services to $\frac{1}{4}$ of the expected maximum difficulty that a high-end CPU can obtain in 24 hours. This means that high-end CPUs should take about six hours to generate a lottery ticket. We benchmarked the performance of our reference implementation of the proof-of-work algorithm on low-end and high-end consumer-grade hardware. An Intel Core2 Quad Q9000 2.00 GHz Penryn CPU from late 2008 performed 1,773.7 PoW iterations per second on one CPU core and 6,165.9 i/s on all four CPU cores. Our high-end machine held an Intel i7-6700K 4.0 GHz Skylake CPU from 2015 and performed 8709.2 i/s on one CPU core and 33,930 i/s on eight cores. Since each iteration is independent probability event, the following formula derives the minimum value v_{th} such that a CPU at S i/s has a 50% chance to compute the result to the PoW in six hours:

$$v_{th} \approx \frac{0.5 \cdot 2^{256}}{6 \cdot 60 \cdot 60 \cdot S} \tag{4}$$

We consider that the i7-6700k represents the upper-end of consumer CPUs and use this formula to recommend $v_{th} = 2^{223.55}$ and $d_{th} = 256 - 224 = 32$ bits. Even with lower-end CPUs, fewer utilized cores, or a higher difficulty, the proof-of-work algorithm will generate a valid ticket eventually. We also suggest halving v_{th} every two years so as to keep pace with a global average increase in computational power. With a sufficiently high value of lottery winners and a sufficiently low value for v_{th}, we believe that our lottery can resist a well-resourced adversary while simultaneously serving computationally-weak innocent parties.

6.4 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 [27]. Closing this leakage is difficult. Arguably the simplest approach is to introduce whitelists or blacklists into common web browsers to prevent known pseudo-TLDs 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 [23] (a small HTTP server library) for networking and the Botan [10] library for most cryptographic operations. We encode all the data structures in JSON and base64-

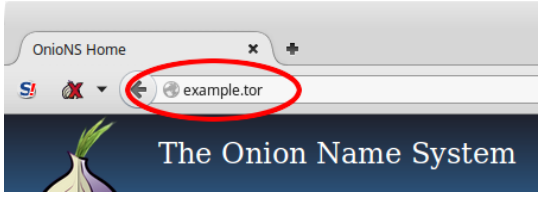

Fig. 5. We load our onionsite, available at "onions55e7yam27n.onion", transparently under the "example.tor" domain. The OnioNS software launches with the Tor Browser.

encode binary data. 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 simply binds to a localhost TCP port and performs our protocols. OnioNS-HS is a 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 6.0.1, a fork of Firefox 45.2.0 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 Fig. 5. The software performed asynchronously and allowed normal browsing to both the Internet and other onion services while it resolved "example.tor".

7.3 Performance

7.3.1 Client-Side Overhead

We measured the client-side overhead of record verification after a Domain Query. We tested the performance of our OnioNS-HS and OnioNS-client software on two machines, M_a and M_b, the Q9000 and i7-6700k CPUs mentioned earlier. We measured the wall-time needed for a client to verify a record or ticket and averaged 200 samples. The results are shown in Table 3.

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

Table 3. The processing time required to verify a record. These measurements also apply to Quorum nodes and mirrors as they also verify tickets and records.

The measurements show that clients can fully validate a record in less than 20 ms even on low-end hardware. Clients must also verify the Merkle tree proof from the mirror to authenticate record and confirm its uniqueness, and they must verify the router descriptors of all Quorum nodes in the consensus in order to authenticate the Merkle tree root hash. The overhead associated for Merkle tree verification is tree height ($log_2(n)$) multiplied by the cost of $\mathcal{H}(x)$. We can estimate the overhead of verifying router descriptors by multiplying the above numbers by $|Q|$. For example, if $|Q| = 127$, then M_b takes 928 ms. Note that this is a one-time cost per registration period.

7.3.2 Communication Overhead

Although Tor is a low-latency network, all OnioNS 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.

In May 2016 we conducted 10,000 measurements of the communication overhead of onion service paths through the Tor network. Each endpoint was hosted on a 24 Mbits AT&T residential connection. Before each measurement, we cleared Tor's state file for both the client and the onion service, forcing both endpoints to generate fresh circuits and select new guard nodes. This allows our measurements to reflect the expected performance in a distributed environment wherein the clients and onion services select their own circuit. We then measured circuit construction to the onion service, the average time of three round trips, and the bandwidth measured during a transfer of a 2 MB payload. We observed that the median circuit construction overhead was 3952 ms, the median circuit latency is 470 ms, and the median bandwidth is 188 KB/s. We have provided supporting box plots of our results in Section A.4.

We expect that a ticket or record would be less than 1 KiB. The mirror must transmit two SHA-256 hashes for each node in the Merkle tree to ensure verifiability. If OnioNS contains Z names, then the expected byte size of a serialized Merkle subtree is approximated by $2 \cdot (49 + e) \cdot \log_2(Z)$ where 49 is the byte size of a base64-encoded SHA-256 hash and e represents markup and other formatting overhead in the transmission and is approximately 98 in our implementation. Assuming 470 ms RTT and 188 KB/s median bandwidth, the client can download up to 99.6 KB of Merkle subtree data to stay within sub-second performance. The expected size of the subtree is 99.6 KB when $Z = 2^{339}$, suggesting that the Domain Query scales to a very large number of names. The client can eliminate circuit construction overhead by connecting to a mirror at startup.

7.3.3 Quorum Scalability

In the Domain Registration protocol (section 5.5.6) potential registrants upload tickets and records to all Quorum nodes. The committment form of these tickets must be uploaded within a short time period (e.g., five minutes) to minimize any information leakage to a potential attacker. Our analysis in Appendix A.1 recommends $|Q| = 127$, so registrants must upload tickets to 127 Quorum nodes within this small window. Considering the aforementioned overhead of 6-hop circuits, in August 2016 we used the same setup as described above to measure the average time required to upload to a variable Quorum size. Our results, shown in Table 4, show that more than a minute is needed to upload to 127 Quorum nodes. Based on these measurements, we recommend a window of 5 minutes, which also provides a margin of error for clock skew.

Quorum Size	Avg. Time (sec)
7	4.62
15	9.24
31	17.95
63	39.84
127	78.41
255	148.89

Table 4. The time required to upload tickets to $2^x - 1$ Quorum nodes using 8 threads to accelerate communication.

We also measured the storage, memory, and processing overhead for Domain Queries for a variable number of records. We generated fixed-size records of ≈ 1 KB each with random names and followed Authenticated Denial-of-Existence algorithm to construct a Merkle tree. Since Table 3 and Appendix A.4 show that the overall time is dominated by the overhead and latency of 6-hop circuits, we removed the network component and measured performance while querying from the same machine. Our results are shown in Table 5.

Records	Disk (MB)	RAM (MB)	CPU (µs)
2^{14}	18	19	7
2^{15}	36	21	32
2^{16}	71	25	145
2^{17}	142	31	257
2^{18}	284	42	634
2^{19}	568	63	1285
2^{20}	1137	108	2272

Table 5. The server-side storage and memory costs to hold the records and the Merkle tree, respectively, and the processing overhead for a Domain Query. The measurements occurred on one thread of the M_b machine with -O3 compiler optimizations.

The table suggests that the processing time is very light and scales logarithmically with respect to the number of records. This time can be further improved through multi-threading as multiple Domain Queries can easily be performed in parallel. The memory and storage costs both scale linearly as expected, as the Merkle tree contains $\mathcal{O}(n)$ nodes. These nodes are relatively cheap to store in RAM. The CPU cost to answer a query is very low, since the Merkle tree is pre-computed.

8 Discussions

In this section we further discuss and compare our work with related works. Our work shares several similarities with Namecoin and OnionDNS. The Domain Query protocol uses a Merkle tree to minimize networking costs, akin to the Simplified Payment Verification (SPV) scheme [19] for thin clients in Bitcoin. Our authenticated denial-of-existence algorithm is analagous to NSEC3 in DNSSEC. However, our system

also provides some features and practical advantages that Namecoin, OnionDNS, and DNSSEC do not.

OnioNS has significantly less overhead than Namecoin: unlike an append-only blockchain, our database can be modified or deleted in-place and mirrors only remember records for non-expired names. Namecoin does not provide any rate-limiting on name registration and is thus vulnerable to land-rushes, denial-of-registration, and domain squatting attacks. This is evident by the severely skewed distribution of wealth in Namecoin; 43.7% of all Namecoins are held by the richest 10 addresses and 90.5% are held by the richest 1,000 addresses. By contrast, 5.3% and 37.5% of all Bitcoins are held by the richest 10 and 1,000 addresses, respectively. [4] Additionally, Namecoin does not provide any mechanisms for authenticated denial-of-existence. Clients must either trust the server or download the blockchain to confirm the claim. We also note that as of June 2016, Namecoin has a hashrate of 440 PH/s, 30.4% of Bitcoin's hashrate [4]. Since we use Bitcoin's blockchain as a beacon to generate $\mathcal{G}(t)$, we consider OnioNS more resistant to computationally-powerful adversaries than Namecoin.

Hashed Authenticated Denial of Existence [13] (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. 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 an 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.

We introduce the *secondaryAddrs* field into the record data structure to achieve load-balancing at a name level, similar to the round-robin scheme in Internet DNS. We note that other tools exist for this purpose such as OnionBalance [9], which uses specialized descriptors to distributes onion service requests across multiple backend Tor instances. We reduce the additional communication overhead by load-balancing requests at the name level. OnioNS also load-balance across its network because clients will query randomly-selected mirrors for tickets or records. If this selection is skewed by consensus weight, then the load will be distributed according to the capabilities of each mirror.

In OnioNS, users trust the Quorum, Tor directory authorities, and Bitcoin during a query. We consider these trustworthy due to Tor's resistance to Sybil attack (Section 6.2), the economic defenses in Bitcoin (Section 6.1), and the low likelihood of Quorum compromise (Appendix A.1 and A.2). As we described in Section 6.2, an OnioNS client can also query past Quorum nodes for additional verification. Transfer and modification records also allow the mirror to prove the authenticity of updates and the user can be warned if the record changes and this proof is not received during a query.

9 Conclusions and Future Work

We have presented the Onion Name System (OnioNS), a decentralized, secure, and privacy-enhanced alternative DNS that maps globally-unique and meaningful names to onion service addresses. OnioNS enables any onion service administrator to anonymously claim a human-readable name for their server and clients to query the system in a privacy-enhanced manner. We introduce mechanisms that let clients authenticate existent names and denial-of-existence claims with minimal overhead. We use Bitcoin as a cryptographically-secure beacon and utilize the existing and semi-trusted infrastructure of Tor. This 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 shows the automatic resolution of domain names, which will allow scaling beyond human-maintained directories. This demonstrates that OnioNS can address 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 [25], the proposed next generation of Tor onion services.

10 Acknowledgments

We would like to thank Roger Dingledine, George Kadianakis, Yawning Angel, Nick Mathewson, and other Tor developers and volunteers within the community for their technical support and commentary on our work. We thank the anonymous reviewers for their insightful comments and suggestions that helped to significantly improve the paper, and also thank Rob Jansen for shepherding our paper. This work was partially supported by NSF grants CNS-1111539, CNS-1314637, CNS-1520552, CNS-1218085, and ACI-1547428.

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] BitInfoCharts, *Crypto-currencies statistics*, https://bitinfocharts.com/, 2016.

[5] Blockchain.info, *Hashrate distribution*, https://blockchain.info/pools, 2016.

[6] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder, *On bitcoin as a public randomness source*, IACR Cryptology ePrint Archive **2015** (2015), 1015.

[7] John Brooks, *Anonymous peer-to-peer instant messaging*, https://github.com/ricochet-im/ricochet, 2016.

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

[9] Donncha O' Cearbhaill, *Onion balance*, https://github.com/DonnchaC/onionbalance, 2016.

[10] Botan Developers, *Botan: Crypto and tls for c++11*, http://botan.randombit.net/, 2016.

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

[12] Ittay Eyal, *The miner's dilemma*, Security and Privacy (SP), 2015 IEEE Symposium on, IEEE, 2015, pp. 89–103.

[13] Internet Engineering Task Force, *Dns security (dnssec) hashed authenticated denial of existence*, https://tools.ietf.org/html/rfc5155, 2008.

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

[15] 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.

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

[17] Trace Mayer, *Bitcoin mining hardware guide*, https://www.bitcoinmining.com/bitcoin-mining-hardware/, 2016.

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

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

[20] 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.

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

[22] 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.

[23] GNU Project, *Microhttpd*, https://www.gnu.org/software/libmicrohttpd/, 2016.

[24] The Tor Project, *Tor metrics*, https://metrics.torproject.org/, 2015.

[25] ———, *Next-generation hidden services in tor*, https://gitweb.torproject.org/torspec.git/tree/proposals/224-rend-spec-ng.txt, 2016.

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

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

[28] 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.

[29] K. T. Wallenius, *Biased sampling: The non-central hypergeometric probability distribution*, Ph.D. Thesis, Stanford University, Department of Statistics. (1963).

[30] Philipp Winter, Roya Ensafi, Karsten Loesing, and Nick Feamster, *Identifying and characterizing sybils in the tor network*, arXiv preprint arXiv:1602.07787 (2016).

A Appendix

A.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 $|Q|$-sized random sample taken from an $|T|$-sized population without replacement, where the population contains $|T| \cdot f_E$ entities that we assume are compromised and colluding. Then the probability that Mallory controls $|E|$ Quorum nodes is given by the hypergeometric distribution, whose probability mass function is shown in Equation 5. Mallory controls the Quorum if either

$|E| > \frac{|Q|}{2}$, or $|E| < \frac{|Q|}{2}$ when the largest agreeing subset of legitimate Quorum nodes has size smaller than $|E|$. The latter scenario is difficult to model theoretically or in simulation, but the probability of the former can be calculated. If all Quorum nodes are selected with equal probability, then $\Pr(|E| > \frac{|Q|}{2})$ is given by the p-value of the hypergeometric test for over-representation, expressed in Equation 6.

$$\Pr(|E|) = \frac{\binom{|T|\cdot f_E}{|E|}\binom{|T|-|T|\cdot f_E}{|Q|-|E|}}{\binom{|T|}{|Q|}} \tag{5}$$

$$\Pr(|E| > \frac{|Q|}{2}) = \sum_{i=\lceil\frac{|Q|}{2}\rceil}^{|Q|} \frac{\binom{|T|\cdot f_E}{i}\binom{|T|-|T|\cdot f_E}{|Q|-i}}{\binom{|T|}{|Q|}} \tag{6}$$

Odd choices for $|Q|$ 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 Fig. 6.

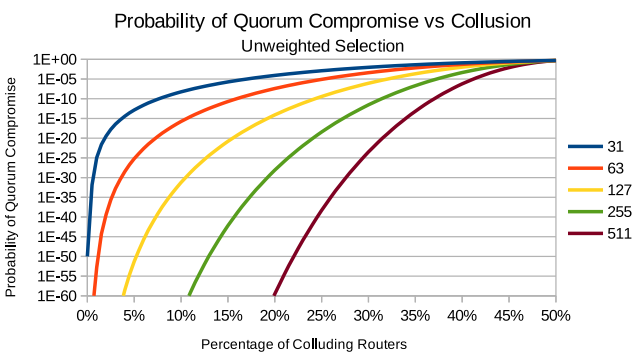

Fig. 6. The values for $\Pr(|E| > \frac{|Q|}{2})$ for Quorum sizes of 31, 63, 127, 255, and 511. All probabilities exceed 0.5 when more than 50 percent of the Tor network is under Mallory's control. We set our population to 4540 routers; the average number of routers with the Fast, Stable, and Running flags across all consensuses in June 2016 [24].

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 Fig. 7. The figure suggests that the Tor network contains a low number of high-end routers and a large number of low-end routers.

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 $|Q|$-sized Quorum from an N-sized population containing

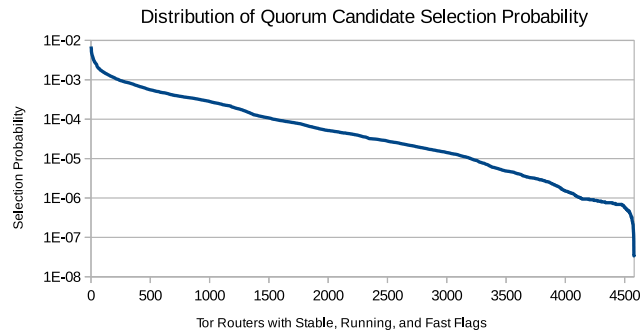

Fig. 7. The normalized distribution of consensus weights of Quorum candidates in June 2016, reflecting the probabilities of inclusion in the Quorum. The distribution may be modelled by an exponential trendline with $R^2 = 0.9884$, but appears slightly super-exponential.

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 compromised routers in a population of size N, is given by Equation 7, and R is the probability that routers outside the lowest x percentile set are compromised. Note that, the probability in Eq. 8 is an approximation of the actual probability of quorum compromise, as we model this problem as sampling balls from an urn without replacement, while the probability of choosing each ball is different. Currently there is no closed-form expression for such a problem. Thus we scale the number of compromised routers with their probabilities of being chosen.

We illustrate the probabilities against discrete values of x and various Quorum sizes in Fig. 8 using $N = 4540$, consistent with the population in Fig. 6.

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

$$\Pr(|E| > \frac{|Q|}{2}) = \sum_{i=\lceil\frac{|Q|}{2}\rceil}^{|Q|} \frac{\binom{K}{i}\binom{N-K}{|Q|-i}}{\binom{N}{|Q|}} \tag{8}$$

In contrast to Fig. 6 which demonstrates that an unweighted selection leads to a high probability of compromise with small levels of collusion, Fig. 8 suggests that biasing Quorum selection by consensus weight provides a strong defense 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 as 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.

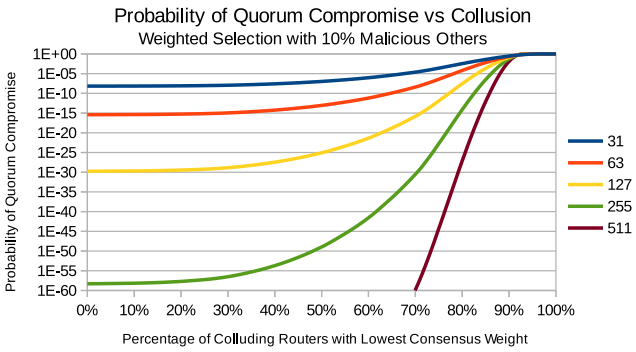

Fig. 8. The values for $\Pr(|E| > \frac{|Q|}{2})$ from Equation 8 for various Quorum sizes. We assume that all routers $\in L(x)$ are under Mallory's control, while routers $\notin L(x)$ have a 10 percent chance of being under Mallory's control.

Small Quorums are also more susceptible to node downtime or denial-of-service attacks. Fig. 8 shows that the choices of $|Q| = 31$ is suboptimal; it is more easily compromised even with low levels of collusion. $|Q| = 63$ is more resistant, but not significantly more so. We therefore recommend $|Q| \geq 127$.

A.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 $|T|$ 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 Fig. 9.

Fig. 9 suggests that although larger values of Δq positively impact security, the choice of $|Q|$ 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 Fig. 9, we further reiterate our recommendation of $|Q| \geq 127$ 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 $|Q|$ and Δq reduces the likelihood of this occurring to near-zero probabilities. We consider this a stronger solution

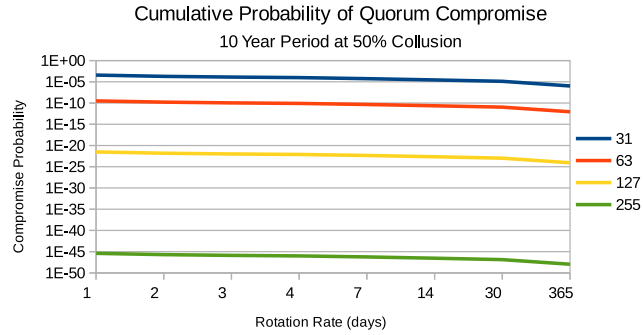

Fig. 9. The cumulative probability that Mallory controls any Quorum at different rotation rates over 10 years at $f_E = 50$ for Quorum sizes 31, 63, 127, and 255. We base these statistics on the probabilities from Fig. 8 at 50 percent collusion.

than introducing countermeasures to specific Quorum-level attacks.

A.3 Lottery Analysis

As mentioned in Section 5.5.6, the land-rush attacks are largely prevented by bootstrapping OnioNS with a reserved set of popular names, including existing Tor onion services. Second, we analyze the effectiveness of the lottery scheme, by studying the DoR attacker's best strategies and outcomes. Obviously, since there are many unpredictable factors in reality (e.g., the number of domain names honest services that will register every day, how everyone registers names, and each registrant's computation power), it is difficult to analyze against all possible scenarios. Thus, we make reasonable assumptions when necessary and try to abstract out the essential idea behind our lottery scheme.

Notations and Assumptions. We assume there is a single well-resourced DoR attacker in the system, who controls $n_a >> 1$ CPUs (or cores). Each attacker CPU's speed is denoted as s_a. There are multiple (m_g) honest onion services which are homogeneous, i.e., each of them has $n_g = 1$ CPU, with the same speed s_g, and only claims one name. The attacker may try to register $m_a \geq 1$ names. Therefore, the total computation power of attacker is expressed in terms of CPU cycles: $P_a = s_a \cdot n_a$, while the total power of honest services is: $P_g = s_g \cdot m_g$. The ticket generation period is Δt. Assume that it takes c_{hash} CPU cycles to do one SHA-256 operation and that the attacker equally spreads its computation power among m_a names. For each of its claimed name, the maximum number of hash operations for the PoW that attacker can carry out during Δt is: $\omega_a = \frac{P_a \cdot \Delta t}{m_a \cdot c_{hash}}$. Let the minimum PoW value v_{min}

he can compute during Δt be equal to ω_a [1], which is used as each of his ticket's weight in the lottery. Similarly, for each honest onion service's ticket, we have $\omega_g = s_g \cdot \Delta t / c_{hash}$. The minimum difficulty every ticket needs to satisfy is denoted as d_{th}.

Theoretical Analysis. Now we consider two extreme cases. First, there are no collisions among attacker's claimed names and honest services' names. Second, that there are a maximum number of collisions possible among those names. We assume that honest service's names are all unique and there can be at most two tickets per name under collision (one from attacker and one from honest service). In the first case, since the attacker's and honest services' tickets generally have different weights and probabilities of being chosen, the lottery process can be modeled as a random sampling process without replacement (urn model) with bias. This can be captured by the Wallenius' noncentral hypergeometric distribution: if an urn contains m_a red balls (attacker's tickets) and m_g white balls (honest services' tickets), totalling $N = m_a + m_g$ balls. Each red ball has the weight ω_a and each white ball has the weight ω_g. The odds ratio is $\omega = \omega_a/\omega_g$. Then, $W \leq N$ balls (winners) are randomly sampled from this urn. Thus, the expected number of attacker's tickets chosen as winner is given by the positive solution μ to the following equation [29]:

$$\frac{\mu}{m_a} + (1 - \frac{W - \mu}{m_g})^{\omega} = 1 \tag{9}$$

Since the above equation does not have closed-form solution, we can solve it using numerical methods.

In the second case, when the number of attacker's tickets is smaller than honest services' ($m_a < m_g$), each attacker's ticket has a collision with an honest service's ticket. For those collided names (m_a), no matter whose ticket is chosen, the other ticket is removed from the lottery. The rest of $m_g - m_a$ have no collisions. Thus, if we regard two tickets under each collided name as a whole red ball, the non-collided tickets as white balls, this is again mapped to a Wallenius' noncentral hypergeometric distribution, with odds ratio being $\omega = (\omega_a + \omega_g)/\omega_g$, $m'_a = m_a$, $m'_g = m_g - m_a$. Now the expected attacker winners is computed as:

$$\mu' = \mu \cdot \frac{\omega_a}{\omega_a + \omega_g}, \tag{10}$$

where μ is the solution to Eq. 9, with m'_a and m'_g as the parameters in it.

When the number of attacker's tickets is larger than honest services' ($m_a > m_g$), each honest service's ticket has a collision with an attacker's ticket. Similarly, we can derive the expected attacker winners in this subcase:

$$\mu' = \mu + (W - \mu) \cdot \frac{\omega_a}{\omega_a + \omega_g}, \tag{11}$$

where μ is again the solution to Eq. 9, with $m'_a = m_a - m_g$, $m'_g = m_g$, and $\omega = \omega_a/(\omega_a + \omega_g)$ as the parameters in it.

Finally, because of the minimum threshold difficulty requirement for every ticket's PoW, if $\omega_a < d_{th}$, that is, if $m_a > \frac{P_a \cdot \Delta t}{d_{th} \cdot c_{hash}}$, all of attacker's tickets will be removed and he will have zero winners in the lottery.

Numerical Results. We did a numerical simulation in Matlab with the following parameter settings as an example. We assume $W = m_g = 50$ (there are 50 honest onion services that win unique names each day), $s_g = s_a = 3$ GHz, the total number of CPUs possessed by the attacker $n_a \geq 1$, $\Delta t = 86400s$ (one day), and $c_{hash} = 512$ CPU cycles to do one SHA-256 operation (according to the benchmark results[2]). Also, attacker's strategy m_a is a variable, which can range from 1 to 1000. We plot the result as the expected number of attacker won tickets v.s. m_a, for both cases (with or without collisions), with different amount of attacker computation powers n_a. It can be seen from Fig. 10 that, the attacker's expected winner size grows with m_a and has a cutoff when its difficulty is less than the threshold. The blue dashed line stands for the power ratio, which is the fraction of the attacker's total computation power to the total power of attacker and honest services'. When m_a is larger than n_a (the point when each of attacker's name is backed by equal computation power to a honest one), the expected attacker winners grow at a decreasing rate, until the cutoff point. If we choose the threshold difficulty level to be a little smaller than what legitimate service's computation power allows (i.e., 1/4 of the maximum difficulty level a legitimate high-end CPU can compute in a 24 hour period, shown in Fig. 10), the maximum expected percentage of attacker's presence in the lottery winners is approximately the power ratio. This is important since the attacker needs to compete with the entire set of honest services' computation power to launch the DoR

1 This is an approximation. To obtain a value less than v, the expected number of hash operations is $2^{256}/v$.

2 https://www.cryptopp.com/benchmarks.html

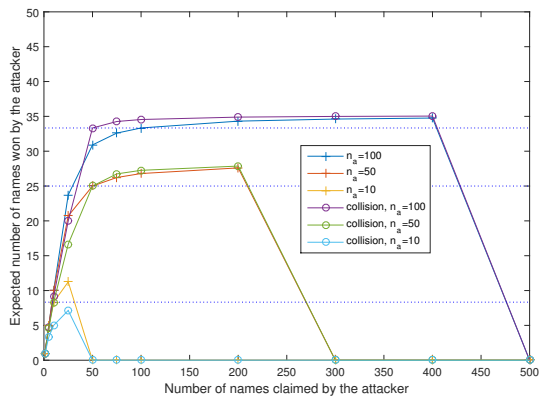

Fig. 10. The expected number of attacker-won tickets in the lottery.

attack. The honest services always have a chance of being selected as winners regardless of the strength of the attacker.

Another observation is that in the name collision case when m_a is small, the expected attacker winners is smaller than the no-collision case, but when m_a is big, it is larger. This is because, when m_a is small, honest onion services also have a chance being selected. Whenever that happens, it removes one ticket from the attacker, thus it reduces the chance of the attacker winning.

Comparing with OnionDNS's domain registration method [26], if the attacker can estimate the number of honest services and their difficulty levels, in OnionDNS the attacker would win many names using the minimum difficulty level since most of them may not have a competitor, and if attacker has a higher difficulty level it wins an individual game deterministically. However, in our design, this is not possible due to the randomization and rate-limiting factor of the lottery, and also the fact that the lottery weight of a ticket is proportional to their computation power dedicated to each ticket, which diminishes exponentially with the difficulty level of the PoW in each ticket. So it is not possible for an attacker gain overwhelming presence in the final set of winners. We consider that the registration schemes in OnionDNS and our work can potentially be complimentary since our lottery scheme is more suitable for a distributed Quorum.

A.4 Communication Overhead of Onion Service Circuits

As we mentioned in Section 7.3.2, we measured the performance of 6-hop onion service circuits. We measured the time required to construct initial circuits to the onion service, the average round-trip time (RTT), and the bandwidth for a 2 MB payload. We performed 10,000 samples and reset Tor's state file on both endpoints each time, forcing each Tor instance to negotiate fresh circuits through the network. We provide box-and-whisker plots of our results in Fig. 11, 12, and 13.

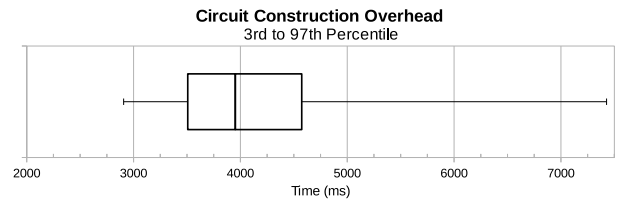

Fig. 11. The median overhead for connecting with an onion service is 3952 ms with a standard deviation of 1794 ms. The whiskers span the 3rd to 97th percentile at 2907 ms and 7353 ms, respectively.

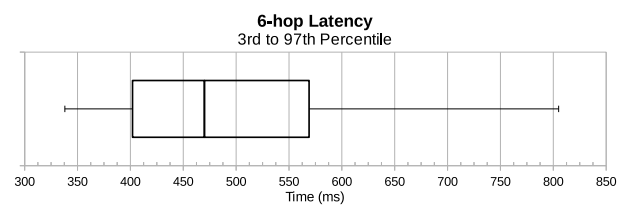

Fig. 12. The median latency of an onion service circuit is 470 ms with a standard deviation of 184 ms. The 3rd and 97th percentile are 338 ms and 805 ms, respectively.

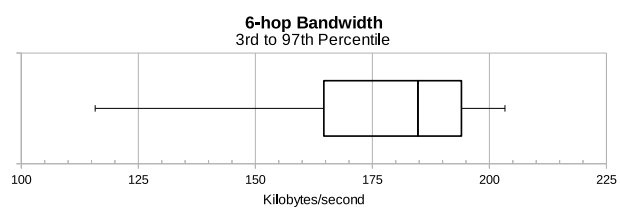

Fig. 13. The median bandwidth for a 2 MB transfer over an onion service circuit is 188 KB/s with a standard deviation of 23 KB/s. The 3rd and 97th percentile are 116 KB/s and 203 KB/s, respectively.