ESGALDNS:
TOR-POWERED DISTRIBUTED DNS
FOR TOR HIDDEN SERVICES

by

Jesse Victors

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Ming Li Major Professor	Dr. Nicholas Flann Committee Member
Dr. Daniel Watson Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2015

CONTENTS

LIST OF FIGURES

CHAPTER 1
REQUIREMENTS

1.1 Assumptions and Threat Model

The design of EsgalDNS on several main assumptions and threat vectors:

- Not all Tor nodes can be trusted. It is already well-known in the Tor community that some Tor nodes are run by malicious operators, curious researchers, experimenting developers, or government organizations. Nodes can be wiretapped, become semi-honest, or behave in an abnormal fashion. However, the majority of Tor nodes are honest and trustworthy: a reasonable assumption considering that Tor's large userbase must make this assumption when using Tor for anonymity or privacy-enhancement purposes.

- For an M-sized set chosen randomly from Tor nodes that have the stable and fast flags, $\lceil \frac{M}{2} \rceil$ or more of them are at least semi-honest.

- The amount of dishonest Tor nodes does not increase in response to the inclusion of EsgalDNS into Tor infrastructure. Specifically, that if an attacker Eve can predict the next set of *quorum* nodes (section 3.3.3) that Eve does not have enough time to make those nodes dishonest. This is a reasonable assumption because regular Tor traffic is far more valuable to an attacker than DNS data is, so penetration of the Tor network would have occurred already if Eve meant to introduce disruption. EsgalDNS data structures are almost all public anyway.

- Adversaries have access to some of Tor inter-node traffic and to portions of general Internet communication. However, attackers do not have not have a global view of Internet traffic; namely they cannot always correlate connections into the Tor network with connections out of the Tor network. This assumption is also made by the Tor community and developers. No attempt is made to defend against a global attacker from either Tor or EsgalDNS.

- Adversaries are not breaking properly-implemented Tor circuits and their modern components, namely TLS 1.2, the AES cipher, ECDHE key exchange, and the SHA2 series of digests, and that they maintain no backdoors in the Botan and OpenSSL implementations of these algorithms.

1.2 Design Principles

Tor's high security environment is challenging to the inclusion of additional capabilities, even to systems that are backwards compatible to existing infrastructure. Anonymity, privacy, and general security are of paramount importance. We enumerate a short list of requirements for any secure DNS system designed for safe use by Tor clients. We later show how existing works do not meet these requirements and how we overcome these challenges with EsgalDNS.

1. The registrations must be anonymous; it should be infeasible to identify the registrant from the registration.

2. Lookups must be anonymous or at least privacy-enhanced; it should not be trivial to determine both a client's identity and the hidden service that the client is requesting.

3. Registrations must be publicly confirmable; all parties must be able to verify that the registration came from the desired hidden service and that the registration is not a forgery.

4. Registrations must be securely unique, or have an extremely high chance of being securely unique such as when this property relies on the collision-free property of cryptographic hash functions.

5. It must be distributed. The Tor community will adamantly reject any centralized solution for Tor hidden services for security reasons, as centralized control makes correlations easy, violating our first two requirements.

6. It must remain simple to use. Usability is key as most Tor users are not security experts. Tor hides non-essential details like routing information behind the scenes, so additional software should follow suite.

7. It must remain backwards compatible; the existing Tor infrastructure must still remain functional.

8. It should not be feasible to maliciously modify or falsify registrations in the database or in transit though insider attacks.

Several additional objectives, although they are not requirements, revolve around performance: it should be assumed that it is impractical for clients to download the entirety or large portions of the DNS database in order to verify any of the requirements, a DNS system should take a reasonable amount of time to resolve domain name queries, and that the system should not introduce any significant load on client computers.

CHAPTER 2

CHALLENGES

2.1 Zooko's Triangle

One of the largest challenges is inherent to the difficulty of designing a distributed system that maintains a correlation database of human-meaningful names in a one-to-one fashion. The problem is summarized in Zooko's Triangle, an influential conjecture proposed by Zooko Wilcox-O'Hearn in late 2001. The conjecture states that in a persistent naming system, only two out of the three following properties can be established: [1]

- Human meaningfulness: the names have a quality of meaningfulness and memorability to the users.
- Securely unique: for any name, duplicates do not exist.
- Distributed: the naming system lacks a central authority or database for allocating and distributing names.

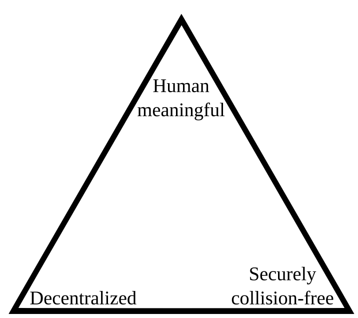

Figure 2.1: Zooko's Triangle.

Tor hidden service .onion domains, PGP keys, and Bitcoin addresses are secure and decentralized but are not human-meaningful; they use the large key-space and the collision-free properties of secure digest algorithms to ensure uniqueness, so no centralized database is needed to provide this property. Tradition domain names on the Clearnet are memorable and provably collision-free, but use a hierarchical structure and central authorities under the jurisdiction of ICANN. Finally, human names and nicknames are meaningful and distributed, but not securely collision-free. [2]

2.2 Communication

CHAPTER 3
SOLUTION

3.1 Overview

I propose a new DNS system for Tor hidden services, which I am calling Esgal Domain Name System. *Esgal* is a Sindarin Elvish noun from the works of J.R.R Tolkien, meaning "veil" or "cover that hides". [3] EsgalDNS is a distributed DNS system embedded within the Tor network on top of the existing Tor hidden service infrastructure. EsgalDNS shares some design principles with Namecoin and its domain names resemble traditional domain names on the clearnet. At a high level, the system is powered at any given time by a randomly-chosen subset of Tor nodes, whose primary responsibilities are to receive new DNS records from hidden service operators, propagate the records to all parties, and save the records in a main long-term data structure. Other Tor nodes may mirror this data structure, distributing the load and responsibilities to many Tor nodes. The system supports a variety of command and control operations including Create, Domain Query, Onion Query, Modify, Move, Renew, and Delete.

3.2 Cryptographic Primitives

Our system makes use of cryptographic hash algorithms, digital signatures, proof-of-work, and a pseudorandom number generator. As the cryptographic data within our system must persist for many years to come, we select well-established algorithms that we predict will remain strong against cryptographic analysis in the immediate future.

- Hash function - We choose SHA-384 for most applications for its greater resistance to preimage, collision, and pseudo-collision attacks over SHA-256, which is itself significantly stronger than Tor's default hidden service hash algorithm, SHA-1. Like

SHA-512, SHA-384 requires 80 rounds but its output is truncated to 48 bytes rather than the full 64, which saves space.

- Digital signatures - Our default method is EMSA-PSS, (EMSA4) a probabilistic signature scheme defined by PKCS1 v2.1 and republished in 2003's RFC 3447, using a Tor node's 1024-bit RSA key with the SHA-384 digest to form the signature appendix. For signatures inside our proof-of-work scheme, we rely on EMSA-PKCS1-v1.5, (EMSA3) defined by 1998's RFC 2315. In contrast to EMSA-PSS, its deterministic nature prevents hidden service operators from bypassing the proof-of-work and brute-forcing the signature to validate the record.

- Proof-of-work - We select scrypt, a password-based key derivation function which is notable for its large memory and CPU requirements during its operation. The scrypt function provides significantly greater resistance to custom hardware attacks and massively parallel computation primarily due to its memory requirements. This limits attackers to the same software implementation and asymptotic cost as legitimate users. [4] We choose scrypt because of these advantages over other key derivation functions such as SHA-256 or PBKDF2.

- Pseudorandom number generation - In applications that require pseudorandom numbers from a known seed, we use the Mersenne Twister generator. In all instances the Mersenne Twister is initialized from the output of a hash algorithm, negating the generator's weakness of producing substandard random output from certain types of initial seeds.

We use the JSON format to encode records and databases of records. JSON is significantly more compact than XML, but retains readability. Its support of basic primitive types is highly applicable to our needs. Additionally, we consider the JSON format safer than byte-level encoding.

3.3 Participants

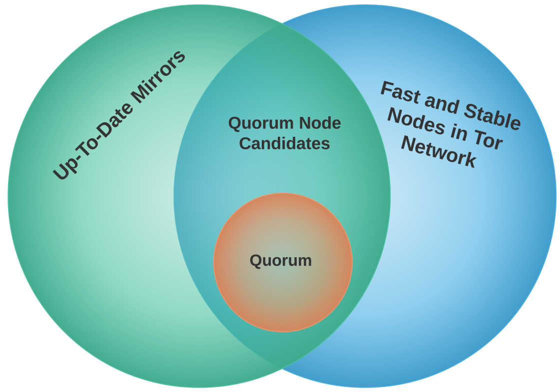

Figure 3.1: There are three sets of participants in the EsgalDNS network: *mirrors*, quorum node *candidates*, and *quorum* members. The set of *quorum* nodes is chosen from the pool of up-to-date *mirrors* who are reliable nodes within the Tor network.

EsgalDNS is a distributed system and may have many participants; any machine with sufficient storage and bandwidth capacity — including those outside the Tor network — can obtain a full copy of all DNS information from EsgalDNS nodes. Inside the Tor network, these participants can be classified into three sets: *mirrors*, quorum node *candidates*, and *quorum* nodes. The last set is of particular importance because *quorum* nodes are the only participants to actively power EsgalDNS.

3.3.1 Mirrors

Mirrors are Tor nodes that have performed a full synchronization (section 3.5.1) against the network and hold a complete copy of all EsgalDNS data structures. This may optionally respond to passive queries from clients, but do not have power to modify any data structures. *Mirrors* are the largest and simplest set of participants.

3.3.2 Quorum Node Candidates

Quorum node *candidates* are *mirrors* inside the Tor network that desire and qualify to become *quorum* nodes. The first requirement is that they must be an up-to-date and complete *mirror*, and secondly that they must have sufficient CPU and bandwidth capabilities to handle the influx of new records and the work involved with propagating these records to other *mirrors*. These two requirements are essential and of equal importance for ensuring that *quorum* node can accept new information and function correctly.

To meet the first requirement, Tor nodes must demonstrate their readiness to accept new records. The naïve solution is to have Tor nodes and clients simply ask the node if it was ready, and if so, to provide proof that it's up-to-date. However, this solution quickly runs into the problem of scaling; Tor has $\approx$ 7000 nodes and $\approx$ 2,250,000 daily users [5]: it is infeasible for any single node to handle queries from all of them. The more practical solution is to publish information to the authority nodes that will be distributed to all parties in the consensus document. Following a full synchronization, a *mirror* publishes this information in the following manner:

1. Let *tree* be its local *AVL Tree*, described in section 3.4.4.
2. Define s as SHA-384($tree$).
3. Encode s in Base64 and truncate to 8 bytes.
4. Append the result to the Contact field in the relay descriptor sent to the authority nodes.

While ideally this information could be placed in a special field set aside for this purpose, to ease integration with existing Tor infrastructure and third-party websites that parse the consensus document (such as Globe or Atlas) we use the Contact field, a user-defined optional entry that Tor relay operators typically use to list methods of contact such as email addresses and PGP keys. EsgalDNS would not be the first system to embed special information in the Contact field; onion-tip.com identifies Bitcoin addresses in the field and then sends shares of donations to that address proportional to the relay's consensus weight.

On weakness with this approach is that because this hash is published in the 00:00 GMT descriptor, an adversary could very easily forge the hash for the 01:00 GMT descriptor and onward and thus broadcast the correct hash without ever performing a synchronization. Combining this hash publication with a Time-based One-time Password Algorithm (TOTP) at a 1 hour time interval.

Of all sets of relays that publish the same hash, if *mirror* m_i publishes a hash that is in the largest set, m_i meets the first qualification to become a quorum node *candidate*. Relays must take care to refresh this hash whenever a new *quorum* is chosen. Assuming complete honesty across all *mirrors* in the Tor network, they will all publish the same hash and complete the first requirement.

The second criteria requires Tor nodes to prove that has sufficient capabilities to handle the increase in communication and processing. Fortunately, Tor's infrastructure already provides a mechanism that can be utilized to prove reliability and capacity; Tor nodes fulfil the second requirement if they have the *fast*, *stable*, *running*, and *valid* flags. These demonstrate that they have the ability to handle large amounts of traffic, have maintained a history of long uptime, are currently online, and have a correct configuration, respectively. As of February 2015, out of the 7000 nodes participating in the Tor network, 5400 of these node have these flags and complete the second requirement.

Both of these requirements can be determined in $\mathcal{O}(n)$ time by anyone holding a recent or archived copy of the consensus document.

3.3.3 Quorum

Quorum are randomly chosen from the set of quorum node *candidates*. The *quorum* perform the main duties of the system, namely receiving, broadcasting, and recording DNS records from hidden service operators. The *quorum* can be derived from the pool of *candidates* by performing by the following procedure, where i is the current day:

1. Obtain a remote or local archived copy of the most recent consensus document, *cd*, published at 00:00 GMT on day $\lfloor \frac{i}{\Delta i} \rfloor$.

2. Extract the authorities' digital signatures, their signatures, and verify *cd* against $PK_{authorities}$.

3. Construct a numerical list, *ql* of quorum node *candidates* from *cd*.

4. Initialize the Mersenne Twister PRNG with SHA-384(*cd*).

5. Use the seeded PRNG to randomly scramble ql.

6. Let the first M nodes, numbered $1..M$, define the *quorum*.

In this manner, all parties — in particular Tor nodes and clients — agree on the members of the *quorum* and can derive them in $\mathcal{O}(n)$ time. As the *quorum* changes every Δi days, *quorum* nodes have an effective lifetime of Δi days before they are replaced by a new *quorum*. Old *quorum* nodes then maintain their *page* (section 3.4.2) as an archive and make it available to future *quorums*.

3.4 Data Structures

3.4.1 Record

A record is a simple data structure issued by hidden service operators to *quorum* members. There are a number of different types of records, each representing a corresponding control operation (section 3.5). However, every record includes a public hidden service key, is self-signed, and contains a full list of all domain names claimed by the hidden service operator. Second-level domains use the .tor TLD (e.g. example.tor) and may be prefixed by sets of label-delimiter pairs to create subdomains. These are collectively known as domain names. This format allows records to be referenced by their central second-level domain names as all corresponding data is encapsulated within the record itself. The details of records, their construction, their transmission, and their application in the system are described in later sections.

3.4.2 Page

A *page* is long-term JSON-encoded textual database held by quorum nodes. It contains five fields, *prevHash*, *recordList*, *consensusDocHash*, *nodeFingerprint*, and *pageSig*.

prevHash

The SHA-384 hash of *prevHash*, *recordList*, and *consensusDocHash* of a previous page, p_{i-1}.

recordList

An array of records, sorted in a deterministic manner.

consensusDocHash

The SHA-384 of *cd*.

nodeFingerprint

The fingerprint of the Tor node, found by generating a hash of the node's public key. This fingerprint is widely used in Tor infrastructure and in third-party tools as a unique identifier for individual Tor nodes.

pageSig

The digital signature, signed with the node's private key, of the preceding fields.

Each *quorum* node has its own *page*. If the nodes in $quorum_{i-1}$ remain online and our assumption the majority are acting honestly, there will exist sets ("clusters") of *pages* that have matching *prevHash*, *recordList*, and *consensusDocHash* fields. Let the choice of p_{i-1} in p_i be the most recent *page* in the chain chosen by the nodes in the largest such cluster. In the event that p_{i-1} or its records to not follow specifications described herein, p_{i-1} should be chosen from the second largest cluster, and so on until p_{i-1} is chosen from the largest cluster that provides a valid *page*.

When a quorum node *candidate* c_j becomes a member of the *quorum*, it constructs an empty *page*. If $i = 0$ then c_j sets *prevHash* to zeros and generates *nodeFingerprint* and *pageSig*. Otherwise then $i > 0$ so *prevHash* is set as the SHA-384 of *prevHash*, *recordList*, and *consensusDocHash* of p_{i-1}. *recordList* is set as an empty array, and *consensusDocHash* and *nodeFingerprint* are both defined. c_j then signs the preceding fields with its private key, saving the result in *pageSig*. Finally, it constructs a one-hop bidirectional Tor circuit to all other *quorum* nodes. These circuits are used for synchronization and must remain alive for the duration of that *quorum*. Overall this creates $\frac{M*(M-1)}{2}$ new TCP/IP links among *quorum* members.

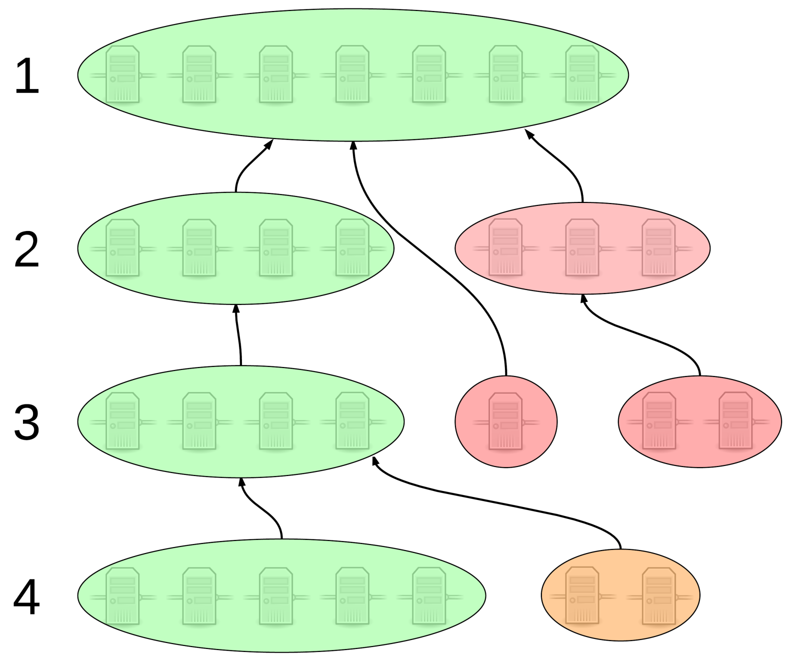

Figure 3.2: An example *page*-chain across four *quorums*. Each *page* contains a references to a previous *page*, forming an distributed scrolling data structure. *Quorums* 1 is semi-honest and maintains its uptime, and thus has identical *pages*. *Quorum* 2's largest cluster is likewise in agreement, but there are three nodes which are acting maliciously together and have changed their *pages*. Node 5 in *quorum* 3 references an old page in attempt to bypass *quorum* 2's records, and nodes 6-7 are colluding with nodes 5-7 from *quorum* 2. Finally, *quorum* 3 has two nodes that acted honestly but did not record new records, so their *page*-chains differ from the others. However, across all four days the largest clusters are honest nodes and thus integrity remains in the *page*-chain.

3.4.3 Snapshot

Similar to a page, a *snapshot* is JSON-encoded textual database held by *quorum* nodes, but unlike pages, *snapshots* are short-term and volatile. They are used for propagating very new records and receiving records from other active *quorum* nodes. Snapshots contain three fields: *originTime*, *recentRecords*, *nodeFingerprint*, and *snapshotSig*.

originTime

Unix time when the snapshot was first created.

recentRecords

A list of records.

nodeFingerprint

The fingerprint of the Tor node.

snapshotSig

The digital signature of the preceding fields, signed using the node's private key.

Snapshots are generated every Δs minutes. At the beginning of one of these intervals, a *quorum* node generates an empty *snapshot*. *OriginTime* is set to the current Unix time, *recentRecords* is an empty array, *nodeFingerprint* is set the same as it is for a *page*, and *snapshotSig* is generated. As records are received, a *quorum* node merges the record into their *snapshot*, as described in section 3.5.2.

3.4.4 AVL Tree

A self-balancing binary AVL tree is used as a local cache of existing records. Its nodes hold references to the location of records in a local copy of the *page*-chain, and it is sorted by alphabetical comparison of second-level domain names. As a *page*-chain is a linear data structure that requires a $\mathcal{O}(n)$ scan to find a record, the $\mathcal{O}(n \log n)$ generation of an AVL tree cache allows lookups of second-level domain names to occur in $\mathcal{O}(\log n)$ time. An AVL tree is generated from a *page*-chain, as described in section 3.5.1.

3.4.5 Hashtable Bitset

A hashtable bitset is a special and highly compact adaptation of a traditional hashtable. Unlike its AVL tree counterpart, the purpose of the hashtable bitset is to prove the non-existence of a record. Demonstrating non-existence is a challenge often overlooked in DNS: even if the DNS records can be authenticated by a recipient, (i.e. an SSL certificate or EsgalDNS self-signed records) a DNS resolver may lie to a client and claim a false negative on the existence of a domain name. Aside from trusting the response of a central authority or a local DNS server, a client cannot easily determine the accuracy of this response without downloading all the records and checking for themselves, but this is impractical in most environments. Asking a central trusted authority or a group of authorities (e.g. the *quorum*) for verification is a simple solution, but these queries introduce additional load upon the authorities. The hashtable bitset allows a set of trusted authorities to publish a digitally signed data structure that allows local resolvers to prove non-existence for any non-existent domain name in $\mathcal{O}(1)$ time on average, and with minimal data sent to the client. We extend the data structure by resolving collisions in a manner that eliminates false negatives and allows the proof of non-existence claims in $\mathcal{O}(\log n)$ in the worst case.

Like an ordinary hashtable, the hashtable bitset maps keys to buckets, but in this application it is only necessary to track the existence of a second-level domain name. Therefore we represent each bucket as a bit, creating a compact bitset of $C * n$ size, where n is the number of existing second-level domain names and c is some constant coefficient. The hashtable bitset records a "1" if a second-level domain name exists, and a "0" if not. In the event of a hash collision, all records that map to that bucket should be added to an array list. Following the construction of the bitset, the array should be sorted alphabetically by second-level domain name and then converted into a Merkle tree. A client can verify non-existence by confirming that the hash of the requested second-level domain points to a 0, or if it points to a 1 and the DNS resolver claims non-existence, the resolver must demonstrate it by sending the client the appropriate section of the Merkle tree, as described in section 3.5.8.

Trusted authorities (e.g. the *quorum* of size M) can divide the bitset into Q sections, digitally sign each section, and digitally sign the root hash of the Merkle tree. This allows a DNS resolver to send a $\frac{C*n}{Q}$-sized section of the bitset and its digital signatures to the client, rather than sending the entire bitmap, which may be larger than $\frac{C*n}{Q}$ for some choices of C, Q, and the size of the signatures. The assembly of these signatures is detailed in 3.5.2.

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

3.5 Operations

3.5.1 Synchronization

EsgalDNS records are public knowledge and any machine may download a complete copy of all data structures that encapsulate records. Once the synchronization is complete, that machine becomes a *mirror* and can be a server to other machines, like BitTorrent or other peer-to-peer networks.

Let i be the current day, Δi be the lifetime of the *quorum*, Alice be the machine becoming a *mirror*, and Bob an existing *mirror*.

1. Alice obtains from Bob his $\min(i, L)$ most recent *pages* in his cached *page*-chain, where L is the lifetime of records.

2. Alice also obtains the SHA-384 hash, h_p, of the concatenation of *prevHash*, *recordList*, and *consensusDocHash* for the *page* used by each *quorum* node for all *quorums* between $i - \min(i, L)$ and i. Note that each h_p is digitally signed by its respective *quorum* node. See section 3.5.2 for details on how this information is available to Bob.

3. Alice downloads the $\frac{\min(i,L)}{\Delta i}$ consensus documents published every Δi days at 00:00 GMT between days $i - \min(i, L)$ and i. Alice may download these documents from Bob, but to lighten the burden on Bob she may also obtain them from any other source. Bob may have compressed these beforehand to save space: very high compression ratios can be achieved under 7zip.

4. The last item that Alice fetches from Bob is the hashtable bitset and the root of the Merkle collision table, which has been signed by all current *quorum* members.

5. Starting with the oldest available consensus document and working forward to day $\lfloor \frac{i}{\Delta i} \rfloor$,

    (a) Alice follows the procedures described in section 3.3.3 to calculate the old *quorum*.

    (b) She confirms that the oldest *page* she received from Bob is held by the largest cluster of agreeing *quorum* nodes.

(c) Alice verifies the validity of the *page* and the records contained within it.

(d) Finally, Alice progresses to the next most recent *page*, repeating the procedure but also verifying that the *prevHash* refers the p_{i-1} she was just examining. This process repeats until all $\min(i, L)$ *pages* have been verified.

6. Alice extracts all records from the now-validated *page*-chain and constructs the AVL tree and the hashtable bitset with its Merkle tree containing the collisions. As second-level domains expire every L rotations of the *quorum*, recent Create, Modify, Move, and Renew operations all act as renewals of the domain name and thus are used by Alice to generated these structures. She should process the records in reverse chronological order because a Delete operation causes immediate expiration of an existing domain.

7. She confirms that the signatures on the sections of the bitset and the signatures on the Merkle root hash check out against her generated copy. If they do not, Bob may have manipulated the data and she may need to ask someone else.

8. Finally, Alice may make the *page*-chain and consensus documents that she downloaded from Bob and the binary hashtable that she constructed available to others. She may also respond to Domain and Onion Queries using the AVL tree. She must perform these actions once Alice becomes a quorum member *candidate*.

3.5.2 Broadcast

Operation records, such as Create, Modify, Move, Renew, and Delete must anonymously transmitted through a Tor circuit to the *quorum* by a hidden service operator. First, the operator uses a Tor circuit to fetch from a *mirror* the consensus document on day $\lfloor \frac{i}{\Delta i} \rfloor$ at 00:00 GMT, which he then uses to derive the current *quorum*. Secondly, he asks the *mirror* for the digitally signed hash of the *page* used by each *quorum* node. Third, he randomly selects two *quorum* nodes from the largest cluster of matching *pages* and sends his record to one of them. For security purposes, the operator should use the same entry node for this transmission that their hidden service uses for its communication. The *quorum* node may reject his record if it is invalid, otherwise it accepts it. Fourth, he constructs a circuit to the second node and polls the node 15 minutes later to determine if it has knowledge of the record. If it does, he can be reasonable sure that the record has been properly transmitted and recorded. If the record is not known the next day, the operator should repeat this procedure to ensure that the record is recorded in the *page*-chain.

Each *quorum* node buffers received records into a *snapshot* and then flushing the snapshot to the rest of the *quorum* every T minutes by performing the following :

Let x be the current propagation iteration and $snap_x$ be the currently-active snapshot filled with records over the last T minutes.

1. Generates a new snapshot, labelled $snap_{x+1}$, sets $originTime$ to the current time, creates *snapshotSig*, and sets $snap_{x+1}$ to be the currently active snapshot for collecting new records.

2. Define an empty array list arr.

3. With each node $q_{k \neq j}$ in the *quorum* using its existing one-hop Tor circuits,

   (a) Sends its $snap_x$ and $< pageSig_{q_j}, nodeFingerprint_{q_j} >$, every $< pageSig_{q_k}, nodeFingerprint_{q_k} >$ it has received so far, and its signatures on the sections of the hashtable bitset and on the Merkle tree root to q_k.

   (b) Receives $s_{x,k}$ and $< pageSig_{q_k}, nodeFingerprint_{q_k} >$ from q_k.

   (c) Archives $< pageSig_{q_k}, nodeFingerprint_{q_k} >$ and add any records that did not exist in $snap_x$ to arr.

4. For any missing $< pageSig_{q_k}, nodeFingerprint_{q_k} >$, it asks a random *quorum* member $q_{i \neq k}$ for q_k's $pageSig_{q_k}$. In this way, it has a list of *page* signatures from all *quorum* nodes.

5. Merges $snap_x$ and the records in arr into its *page* and regenerates *pageSig*.

6. Updates its AVL tree, hashtable bitset, and Merkle collision tree, and regenerates the signatures on the bitset and on the Merkle root tree.

7. Increments x.

This process is illustrated in figure 3.3.

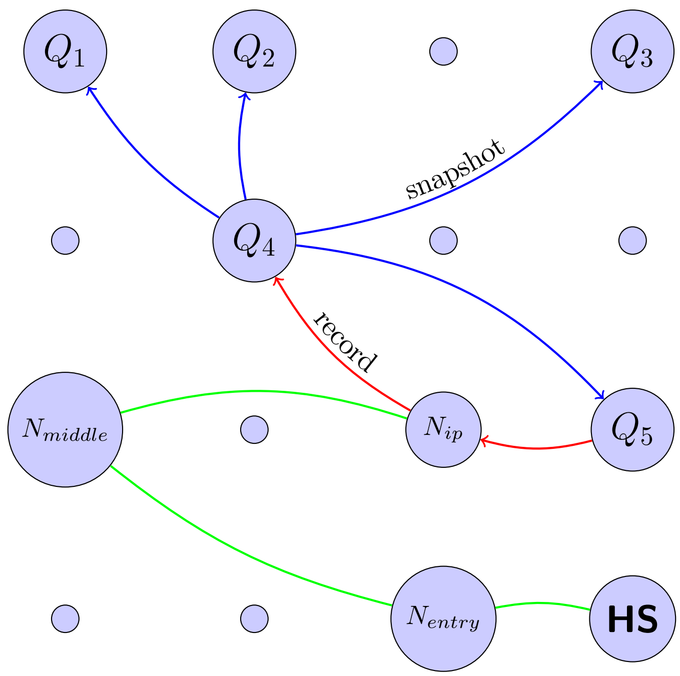

Figure 3.3: The hidden service operator uses his existing circuit (green) to inform *quorum* node Q_4 of the new record. Q_4 then distributes it via *snapshots* to all other *quorum* nodes. Each records it in their own *page* for long-term storage. The operator also confirms from a randomly-chosen *quorum* node Q_5 that the record has been received.

3.5.3 Create

Any hidden service operator may claim any second-level domain name that is not currently in use. Since there is no central authority in a distributed system from which to purchase a domain name, it is necessary to implement a system that introduces a cost of ownership. This fulfils three main purposes:

1. Thwarts potential flooding of the system with operational records.

2. Introduces a cost of investment that encourages the availability of hidden services.

3. Makes domain squatting more difficulty, where someone claims one or more second-level domains on a whim for the sole purpose of denying them to others. As hidden service operators typically remain anonymous, it is difficulty for one to contact them and request relinquishing of a domain, nor is there a central authority to force relinquishing through a court order or other formal means.

Therefore we incorporate a proof-of-work scheme that makes registration computationally intensive but is also easily verified by anyone. A Create record consists of nine components: *type*, *nameList*, *contact*, *timestamp*, *consensusHash*, *nonce*, *pow*, *recordSig*, and *pubHSKey*. Let the variable *central* consist of all fields except *recordSig* and *pow*. Fields that are optional are blank unless specified, and all fields are encoded in base64, except for *nameList*, *contact*, and *timestamp*, which are encoded in standard UTF-8. These are defined in Table 3.5.3.

Field	**Required?**	**Description**
type	Yes	A textual label containing the type of record. In this case, *type* is set to "Create".

nameList	Yes	An array list of up to 24 domain names. Each domain name consists of one or more textual labels prefixing the .tor TLD, delimited by dots. Thus a second-level domain would be example.tor. Domain names can point to other domain names with either .tor or .onion TLDs; this is similar to the Clearnet DNS where domains can be chained but eventually resolve to an IP address. There can be up to eight name-separator pairs prefixing the TLD, and each name can be up to 32 characters long. Domain names must use a second-level domain listed in this field so that hidden service operators cannot claim subdomains on a base domain name that they do not own.
contact	No	The fingerprint of the HS operator's PGP key, if he has one. If the fingerprint is listed, clients may query a public keyserver for this fingerprint, obtain the operator's PGP public key, and contact him over encrypted email.
timestamp	Yes	The UNIX timestamp of when the operator created the registration and began the proof-of-work to validate it.
consensusHash	Yes	The SHA-384 hash of the morning's consensus document at the time of registration. This is a provable timestamp, since it can be matched against archives of the consensus document. Quorum nodes will not accept registration records that reference a consensus document more than 48 hours old.
nonce	Yes	Four bytes that serve as a source of randomness for the proof-of-work, described below.

pow	Yes	16 bytes that demonstrate the result of the proof-of-work.
recordSig	Yes	The digital signature of all preceding fields, signed using the hidden service's private key.
pubHSKey	Yes	The public key of the hidden service. If the operator is claiming a subdomain of any depth, this key must match the *pubHSKey* of the top domain name.

Table 3.1: Fields in the Create record. Every record is self-signed and requires the solving of proof-of-work before it is valid.

A record is made valid through the completion of the proof-of-work process. The hidden service operator must find a *nonce* such that the SHA-384 of *central*, *pow*, and *recordSig* is $\leq 2^{\text{difficulty}}$, where *difficulty* specifies the order of magnitude of the work that must be done. The *difficulty* is doubled every 1460 days. For each *nonce*, *pow* and *recordSig* must be regenerated, which effectively forces the computation to be performed by a machine owned by the HS operator. When the proof-of-work is complete, the valid and complete record is represented in JSON format for transmission to the *quorum*.

3.5.4 Modify

If a hidden service operator wishes to update his registration with more current information, he can broadcast a Modify record. The Modify record has identical fields to Create, but *type* is set to "Modify". The operator creates a Modify record with the corrected information and then sends it to the *quorum*. The Modify operation renews the ownership of second-level domain names, so a record of the domain name must already exist in the *page*-chain and be less than L days old. Once received, *quorum* nodes update the leaf in their AVL trees with the modified record. Modify records have a difficulty of $\frac{\text{difficulty}_{Create}}{4}$.

3.5.5 Move

A Move record may be issued if a hidden service operator wishes to transfer second-level domain names in a record to another owner. Move records have all the fields of a Create record, have their *type* is set to "Move", and contain one additional field: the public key of the new owner. Like Modify records, Move records also renew second-level domain names, so they must already exist in the *page*-chain. Move records also have a difficulty of $\frac{\text{difficulty}_{Create}}{4}$.

3.5.6 Renew

As second-level domain names expire every L days because records older than L are not fetched by *mirrors*, Renew records must be reissued periodically to ensure that the domain names remain in the *page*-chain. Renew records are identical to Create records, except that *type* is set to "Renew" and, like the Modify and Move records, the difficulty is $\frac{\text{difficulty}_{Create}}{4}$.

3.5.7 Delete

Delete records are useful if a hidden service operator wishes to relinquish ownership rights over second-level domain names, (and all their subdomains) or if they consider their private key compromised. Delete records are also identical to Create records, but they have *type* is set to "Delete" and any second-level domain names and subdomains are instantly purged from the caches in all *mirror* nodes and thus become available to others. There is no difficulty associated with Delete records, so they can be issued instantly.

3.5.8 Domain Query

When a Tor user Alice wishes to visit a .tor domain name, her client software must resolve the domain to determine what hidden service she should connect to. As records contain a list of domain names including all subdomains under second-level domains, Alice needs to recursively obtain records until she is led to a .onion TLD. At that point, she can begin the traditional hidden service lookup. This all occurs behind-the-scenes, so from Alice's perspective sub.example.tor takes her to a webpage just like it would for other TLDs in the Clearnet DNS system. Of course, if Alice uses another TLD other than .tor, her web browser should query the Clearnet DNS, otherwise it performs a Domain Query in the EsgalDNS system.

At startup, Alice fetches a copy of the consensus document on day $\lfloor \frac{i}{\Delta i} \rfloor$ at 00:00 GMT, determines the current *quorum*, and builds a circuit to any *mirror* node $resolver$. In practice this *mirror* should be a quorum node *candidate*, but Alice can choose her resolver. When she wishes to resolve a domain name, she asks *resolver* for the records that recursively resolve the .tor domain to the .onion domain. There are three verification levels that Alice can request, each giving Alice progressively more verification that the record she received is authentic, unique, and trustworthy.

At verification level 0, (the default) *resolver* first checks the hashtable bitset to confirm that the record exists. If it does, it queries its AVL tree to find the latest Create, Modify, Move, or Renew record that contains her requested domain name. If the record does not exist, *resolver* returns to Alice the signed sections of the bitset and the signed root of the Merkle collision tree. Secondly, Alice either confirms that the record does not exist, or if so, she verifies *recordSig*, the self-signature of the record's fields under *pubHSKey*. Secondly, Alice confirms that the proof-of-work checks out and meets the expected difficulty level. Third, if the record does not resolve the domain name into a .onion TLD but instead points to a second .tor domain, she can again query *resolver* for that domain name. This repeats up to eight times until Alice has recursively resolved the original domain name. Once this occurs, she generates the hidden service .onion address by converting *pubHSKey* in the final record to PKCS.1 DER encoding, generating the SHA-1 hash, converting the result to base58, and truncating to 16 characters. She can then look up the .onion in the original manner and pass it the original domain name. The lookup fails if the service has not published a recent hidden service descriptor to the distributed hash table. End-to-end verification is complete when *pubHSKey* can be successfully used to encrypt the hidden service cookie and the service proves that it can decrypt *sec* as part of the hidden service protocol.

At verification level 1, the protocol follows everything from level 0, except that *resolver* also returns the *page* that contained each record and any 7zip-compressed consensus documents that Alice needs to resolve past *quorums*. With these documents Alice can confirm that the *page* is authentic to at least one *quorum* node.

Verification level 2 is almost identical to level 1, but Alice also receives the digitally signed hashes of the *page* held by each *quorum* node. This demonstrates width verification, as Alice can confirm that the *page* is in the largest cluster of *quorum* nodes that are all on the same *page*. It is impractical for Alice to download all *pages* to verify the uniqueness and trustworthiness of a returned record, so Alice can fetch minimal additional information and rely on her trust in the Tor network (in particular the *quorum*) to confirm this for her.

Full trust can be achieved for herself if Alice performs a full synchronization against *resolver* and then becomes her own resolver. This allows her to perform verification in width and depth, and she can see for herself that the second-level domain is unique and trustworthy, at least relative to the *quorum* that it was broadcasted to.

3.5.9 Onion Query

An Onion Query can be issued by Alice to *resolver* to find the second-level domains that directly resolve to a given .onion name. This is analogous to a reverse DNS lookup on the Clearnet DNS. In advance and for this purpose, *resolver* generates a trie data structure of .onion addresses in advance with leafs pointing to the latest record. Alice can then verify the returned domain name by issuing a Domain Query at any verification level, which should result in the same .onion address that she originally requested.

3.6 Examples and Structural Induction

3.6.1 Base Case

In the most trivial base case of a single quorum node *candidate* c_1, a hidden service Bob, and a Tor client Alice, the procedures are relatively simple. On day_0, c_1 generates an

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


Figure 3.4: A sample empty *page*, $p_{1,1}$, encoded in JSON and base64.

initial page $p_{1,1}$ containing no records and signs $p_{1,1}$, but does not accept records for this initial page. $p_{1,1}$ appears in Figure 3.4.

On day$_1$, c_1 examines its database of page-chains and generates a new page, $p_{2,1}$, that references $p_{1,1}$, a chain with 0 references, the most in the database. The hidden service *Bob* hashes the consensus document, generating *T/q7q052MgJGLfH1mBGUQSF YjwVn9VvOWBoOmevPZgY=* which is then fed into the Mersenne Twister to scramble the list of *candidate* nodes. Since c_1 is the only *candidate*, he is chosen a member of the *quorum*. Bob then builds a circuit to c_1, and sends him a registration record, r_{reg}, which appears in Figure 3.5.

c_1 can continue to accept and insert records in this way, but if r_{reg} is the only one that c_1 receives, at the next 15 minute mark c_1 will attempt to propagate this snapshot to other *quorum* nodes. However, as c_1 is the only *quorum* node, that step is not necessary here. c_1 then adds r_{reg} into its page, creating $p_{1,1}$, shown in Figure 3.6.

This record $\rightarrow$ snapshot $\rightarrow$ page merge process continues for any new records, but assuming r_{reg} is the only record received that day, $p_{1,1}$ will not change following the end of day$_1$. On day$_2$, c_1, again a *quorum* member, will build a page $p_{1,2}$ that links to $p_{1,1}$, the latest page in the chain with the most links, now 2. Generally speaking, on day day$_n$ c_1 will select $p_{1,n-1}$, as there is no other choice. It alone listens for new records, rejects new registrations if there is a name conflict, and ensure the validity of the entire page-chain

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


Figure 3.5: Sample registration record from a hidden service, encoded in JSON and base64. The "sub.example.tor" → "example.tor" → "exampleruyw6wgve.onion" references can be resolved recursively.

database. The Tor client Alice, wishing to contact the hidden service Bob, may query c_1 for "example.tor" and c_1 returns r_{reg}. Alice can then confirm the validity of r_{reg} herself, follow "example.tor" to "exampleruyw6wgve.onion", and finally perform the traditional hidden service lookup.

3.6.2 First Expansion

Extending the example to two *candidates* c_1 and c_2, a mirror m_1, a hidden service Bob, and a Tor client Alice, the purpose of the *snapshot* and *Merkle Tree* data structures become more clear. This is illustrated in Figure 3.8. As before, on day$_0$, c_1 and c_2 both generate and sign initial empty pages but do not accept records. On day$_1$ however, c_1 and c_2 can both publish hashes of their empty *Merkle Tree* databases. Since there are no *pages*

```
{
    "prevHash": 0,
    "recordList": [
        {
            "names": {
            "example.tor": "exampleruyw6wgve.onion",
            "sub.example.tor": "example.tor"
        }
        "contact": "AD97364FC20BEC80",
        "timestamp": 1424045024,
        "consensusHash": "uU0nuZNNPgilLlLX2n2r+sSE7+N6U4DukIj3rOLvzek=",
        "nonce": "AAAABw==",
        "pow": "4I4dzaBwi4AIZW8s2m0hQQ==",
        "recordSig":    "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/
            kBEpyXRSopzjoFs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh+
            QOnKl0fKBN7fqowjkQ3ktFkR0Vuox9WrrbNTMa4+
            up0Np52hlbKA3zSRz4fbR9NVlh6uuQ=",
        "pubHSKey": "MIGhMA0GCSqGSIb3DQEBAQUAA4GPADCBiwKBgQDE7CP/
            kgwtJhTTc4JpuPkvA7Ln9wgc+
            fgTKgkyUp1zusxgUAn1c1MGx4YhO42KPB7dyZOf3pcRk94XsYFY1ULkF2+
            tf9KdNe7GFzJyMFCQENnUcVXbcwLH4vAeiGK7R/nScbCbyc9LT+
            VE1fbKchTL1QzLVBLqJTxhR+9YPi8x+QIFAdZ8BJs="
        }
    ],
    "consensusDocHash": "T/q7q052MgJGLfH1mBGUQSFYjwVn9VvOWBoOmevPZgY=",
    "nodeFingerprint": "2FC06226AE152FBAB7620BB107CDEF0E70876A7B",
    "pageSig": "KO7FXtoTJmxceJYlW202c0WwRGRyU9m99IskcL9yv/
        wFQ4ubzbjVs8LQzwQub9kDJ8Htpc9rRZvneRRbusFv1nvaeJw+WgRt+
        Tck0uapndHKYaQcK3XTIFYdmT1lLm7QxSKjnIxgBkwKT0QWdGLUhuRgGe5CXmqrPeDfU
        /gsgLs="
}
```


Figure 3.6: c_1's page, containing a single registration record.

to reference aside from the initial base *pages*, c_1 and c_2 should both be in agreement and m_1, Alice, and Bob can all see that they are *candidates* because their published hashes are in the majority. However, if c_2 acts maliciously and causes the rare event that the majority is evenly split, two *quorums* will be generated, which c_1 and c_2 are each a part of. However, this unlikely scenario does not change the behaviour of the system and everything operates as before. In either case, Bob sends to $c_{1\leq j\leq 2}$ his Registration record. As before, c_j adds

```
{
    "originTime": 1424042032,
    "recentRecords: [
        {
            "prevHash": 0,
            "recordList": 0,
            "consensusDocHash": "uU0nuZNNPgilLlLX2n2r+sSE7+
                N6U4DukIj3rOLvzek=",
            "nodeFingerprint": "2FC06226AE152FBAB7620BB107CDEF0E70876A7B",
            "pageSig":  "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/
                kBEpyXRSopzjoFs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh
                +QOnKl0fKBN7fqowjkQ3ktFkR0Vuox9WrrbNTMa4+
                up0Np52hlbKA3zSRz4fbR9NVlh6uuQ="
        }
    ],
    "nodeFingerprint": "2FC06226AE152FBAB7620BB107CDEF0E70876A7B",
    "snapshotSig": "FUgZLuFUbh0E0AKbrl1k7/4O7ucPvlr7QFkG1i9/mNFgyH/6TwNQ+
        d2Gsch/9FaN6ZjyHAnvjmSpRRSngR0UD20FwpAZ1vCVA0qO2yDZeuBd6DiNS
        kkdSueRHOF7OD95Rb04JmAk1jXjEgFb+BH3hUH54ZEaqlJvQ8tBQJ7YtAc="
}
```


Figure 3.7: Sample snapshot from c_j, containing one registration record r_{reg} from a hidden service.

the record to its page and at the next 15 minute mark sends the *snapshot* out to the other *quorum* nodes. The snapshot is illustrated in Figure 3.7. Then c_1 and c_2 both have Bob's record, they both hold two pages, and they are in agreement as to the *pages* and data they are using.

m_1 mirrors the *quorum*, so Alice can query m_1 for a name and m_1 returns the Registration or Ownership Transfer record, whichever appeared later. day$_1$ the *page*-chain has four links: the blank links in the two origin *pages* and the two equal links from the two day$_1$ *pages* to the origin *pages*. Therefore the *quorum* on day$_2$ (again c_1 and c_2) generate *pages* that reference the day$_1$ page, which the same for c_1 and c_2. As the days progress, if c_1 and c_2 ever disagree about the *page* to use and the *quorum* is therefore evenly split, by the rules of *page* selection the following day's *quorum* will choose whichever *page* contains the most records, or c_1's *page* if they are both equally sized.

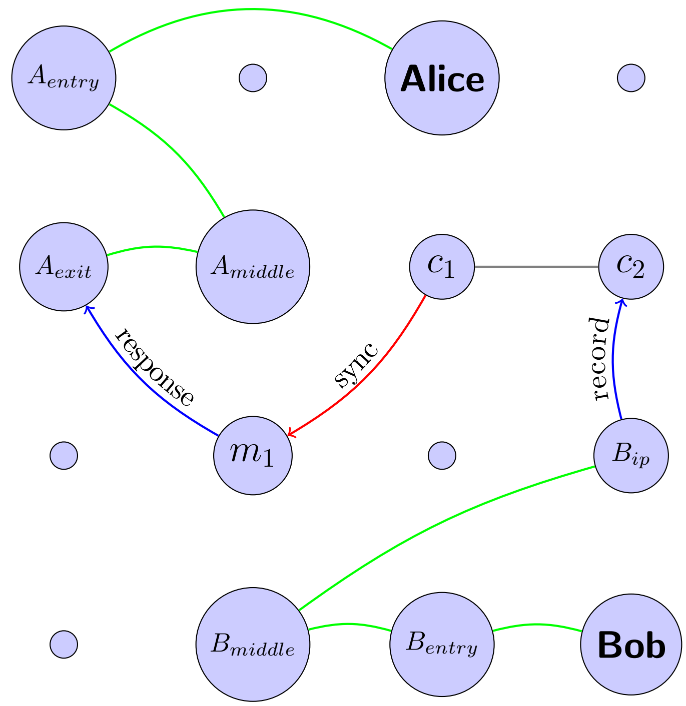

Figure 3.8: The hidden service operator Bob anonymously sends a record to the *quorum* (c_1 and c_2), informing them about his domain name. A node m_1 mirrors the *quorum*, which Alice anonymously queries for Bob's domain name.

3.6.3 General Example

Generally, there are $N \in \mathbf{Z}$ *candidate* nodes $c_{1..N}$, a *quorum* $q_{1..M}$ of size $M \in \mathbf{Z}$, $H \in \mathbf{Z}$ hidden services $hs_{1..H}$, and $C \in \mathbf{Z}$ clients $c_{1..C}$. The $c_{1..N}$ nodes publish the hashes of their *Merkle Tree* structures and all parties can confirm that they remain *candidate* nodes as long as their hashes are in the majority. In the unlikely scenario (the chance, assuming random behavior, is $\frac{1}{2}^{\frac{N}{2}}$) that the majority is evenly split, M becomes twice as large as usual. A hidden service $hs_{1 \leq j \leq H}$ uploads records to a *quorum* node $q_{1 \leq k \leq M}$. Every $q_{1 \leq l \leq M}$ shares *snapshots* every 15 minutes with all other $q_{1 \leq p \neq l \leq M}$, so that all *quorum* node contains the record from $hs_{1 \leq j \leq H}$. These records are saved long-term in *pages*, and every $q_{1 \leq k \leq M}$ knows and can verify the *pages* used by all *quorum* members. Assuming perfect behavior and consistent uptime, these *pages* should always be the same. In the event that they diverge, the rules of the network and *page* selection dictate how to select the "best" *page*. Any machine may become a *mirror* by synchronizing against the *quorum* and fetching all *pages*. The $c_{1 \leq y \leq C}$ can then query that *mirror* for names and perform deep verification on the response. As all names link to either other .tor or .onion names and eventually lead to .onion names, any client can resolve a name into a hidden service .onion address and perform the hidden service lookup in the traditional manner.

CHAPTER 4
ANALYSIS

We have designed EsgalDNS to meet our original design goals. Assuming that Tor circuits are a sufficient method of masking one's identity and location, hidden service operators can perform operations on their records anonymously. Likewise, a Tor circuit is also used when a client lookups a .tor domain name, just as Tor-protected DNS lookups are performed when browsing the Clearnet through the Tor Browser. EsgalDNS records are self-signed and include the hidden service's public key, so anyone — particularly the client — can confirm the authenticity (relative to the authenticity of the public key) and integrity of any record. This does not entirely prevent Sybil attacks, but this is a very hard problem to address in a distributed environment without the confirmation from a central authority. However, the proof-of-work component makes record spoofing a costly endeavour, but it is not impossible to a well-resourced attacker with sufficient access to high-end general-purpose hardware.

Without complete access to a local copy of the database a party cannot know whether a second-level domain is in fact unique, but by using an existing level of trust with a known network they can be reasonably sure that it meets the unique edge of Zooko's Triangle. Anyone holding a copy of the consensus document can generate the set of *quorum* node and verify their signatures. As the *quorum* is a set of nodes that work together and the *quorum* is chosen randomly from reliable nodes in the Tor network, EsgalDNS is a distributed system. Tor clients also have the ability to perform a full synchronization and confirming uniqueness for themselves, thus verifying that Zooko's Triangle is complete. Hidden service .onion addresses will continue to have an extremely high chance of being securely unique as long the key-space is sufficiently large to avoid hash collisions.

Just as traditional Clearnet DNS lookups occur behind-the-scenes, EsgalDNS Domain

Queries require no user assistance. Client-side software should filter TLDs to determine which DNS system to use. We introduce no changes to Tor's hidden service protocol and also note that the existence of a DNS system introduces forward-compatibility: developers can replace hash functions and PKI in the hidden service protocol without disrupting its users, so long as records are transferred and EsgalDNS is updated to support the new public keys. We therefore believe that we have met all of our original design requirements.

4.1 Security

4.1.1 Quorum-level Attacks

The quorum nodes hold the greatest amount of responsibility and control over EsgalDNS out of all participating nodes in the Tor network, therefore ensuring their security and limiting their attack capabilities is of primary importance.

Malicious Quorum Generation

If an attacker, Eve, controls some Tor nodes (who may be assumed to be colluding with one another), the attacker may desire to include their nodes in the quorum for malicious manipulation, passive observation, or for other purposes. Alternatively, Eve may wish to exclude certain legitimate nodes from inclusion in the quorum. In order to carry out either of these attacks, Eve must have the list of qualified Tor nodes scrambled in such a way that the output is pleasing to Eve. Specifically, the scrambled list must contain at least some of Eve's malicious nodes for the first attack, or exclude the legitimate target nodes for the second attack. We initialize Mersenne Twister with a 384-bit seed, thus Eve can find k seeds that generates a desirable scrambled list in 2^{192} operations on average, or 2^{384} operations in the worst case. The chance of any of those seeds being selected, and thus Eve successfully carrying out the attack, is thus $\frac{2^{384}}{k}$.

Eve may attempt to manipulate the consensus document in such a way that the SHA-384 hash is one of these k seeds. Eve may instruct her Tor nodes to upload a custom status report to the authority nodes in an attempt to maliciously manipulate the contents of the consensus document, but SHA-384's strong preimage resistance and the unknown state and number of Tor nodes outside Eve's control makes this attack infeasible. As of the time of this writing, the best preimage break of SHA-512 is only partial (57 out of 80 rounds in 2^{511} time [6]) so the time to break preimage resistance of full SHA-384 is still 2^{384} operations. This also implies that Eve cannot determine in advance the next consensus document, so the new quorum cannot be predicted. If Eve has compromised at least some of the Tor authority nodes she has significantly more power in manipulating the consensus document for her own purposes, but this attack vector can also break the Tor network as a whole and is thus outside the scope of our analysis. Therefore, the computation required to maliciously generate the quorum puts this attack vector outside the reach of computationally-bound adversaries.

EsgalDNS and the Tor network as a whole are both susceptible to Sybil attacks, though these attacks are made significantly more challenging by the slow building of trust in the Tor network. Eve may attempt to introduce large numbers of nodes under her control in an attempt to increase her chances of at least one of the becoming members of the *quorum*. Sybil attacks are not unknown to Tor; in December 2014 the black hat hacking group LizardSquad launched 3000 nodes in the Google Cloud in an attempt to intercept the majority of Tor traffic. However, as Tor authority nodes grant consensus weight to new Tor nodes very slowly, despite controlling a third of all Tor nodes, these 3,000 nodes moved 0.2743 percent of Tor traffic before they were banned from the Tor network. The Stable and Fast flags are also granted after weeks of uptime and a history of reliability. As nodes must have these flags to be qualified as a *quorum candidate*, these large-scale Sybil attacks are financially demanding and time-consuming for Eve.

4.1.2 Non-existence Forgery

As we have stated earlier, falsely claiming a negative on the existence of a record is a problem overlooked in other domain name systems. One of the primary challenges with this approach is that the space of possible names so vast that attempting to enumerate and digitally sign all names that are not taken is highly impractical. Without a solution, this weakness can degenerate into a denial-of-service attack if the DNS resolver is malicious towards the client. Our counter-measure is the highly compact hashtable bitset with a Merkle tree for collisions. We set the size of the hashtable such that the number of collisions is statistically very small, allowing an efficient lookup in $\mathcal{O}(1)$ time on average with minimal data transferred to the client.

4.1.3 Name Squatting and Record Flooding

An attacker, Eve, may attempt a denial-of-service attack by obtaining a set of names for the sole purpose of denying them to others. Eve may also wish to create many name requests and flood the *quorum* with a large quantity of records. Both of these attacks are made computationally difficult and time-consuming for Eve because of the proof-of-work. If Eve has access to large computational resources or to custom hardware she may be able to process the PoW more efficiently than legitimate users, and this can be a concern.

The proof-of-work scheme is carefully designed to limit Eve to the same capabilities as legitimate users, thus significantly deterring this attack. The use of scrypt makes custom hardware and massively-parallel computation expensive, and the digital signature in every record forces the hidden service operator to resign the fields for every iteration in the proof-of-work. While the scheme would not entirely prevent the operator from outsourcing the computation to a cloud service or to a secondary offline resource, the other machine would need the hidden service private key to regenerate *recordSig*, which the operator can't reveal without compromising his security. However, the secondary resource could perform the scrypt computations in batch without generating *recordSig*, but it would always perform more than the necessary amount of computation because it would could not generate the SHA-384 hash and thus know when to stop. Furthermore, offloading the computation would still incur a cost to the hidden service operator, who would have to pay another party for the consumed computational resources. Thus the scheme always requires some cost when claiming a domain name.

4.2 Performance

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

4.2.1 Load

demand on participating nodes to be determined...

Unlike Namecoin, EsgalDNS' *page*-chain is of L days in maximal length. This serves two purposes:

1. Causes domain names to expire, which reduced the threat of name squatting.
2. Prevents the data structure from growing to an unmanageable size.

4.3 Reliability on Unreliable Hosts

Tor nodes have no reliability guarantee and may disappear from the network momentarily or permanently at any time. Old *quorums* may disappear from the network without consequence of data loss, as their data is cloned by current *mirrors*. So long as the *quorum* nodes remain up for the Δi days that they are active, the system will suffer no loss of functionality. Nodes that become temporarily unavailable will have out-of-sync *pages* and will have to fetch recent records from other *quorum* nodes in the time of their absence.

CHAPTER 5

CONCLUSION

EsgalDNS is a distributed DNS system inside the Tor network. The system maps human-meaningful .tor domain names to traditional Tor .onion addresses. It exists on top of current Tor hidden service infrastructure and increases usability of hidden services by abstracting away the base58-encoded hidden service addresses. EsgalDNS is powered by a randomly chosen set of Tor nodes, whose work can be verified by any party with a copy of Tor's consensus document. Records are self-signed and can be publicly verified. If accepted by the Tor community, EsgalDNS will be a valuable abstraction layr that will significantly improve the usability and popularity of Tor hidden services.

REFERENCES

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

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

[3] D. Willis, "Hiswelk's sindarin dictionary," http://www.jrrvf.com/hisweloke/sindar/online/sindar/dict-sd-en.html, edition 1.9.1, lexicon 0.9952, accessed 16-February-2015.

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

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

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