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

PUBLIC ABSTRACT

Jesse M. Victors

The Tor network is a second-generation onion routing system that aims to provide anonymity, privacy, and Internet censorship resistance to its users. In recent years it has grown significantly in response to revelations of national and global electronic surveillance, and remains one of the most popular and secure anonymity network in use today. Tor is also known for its support of anonymous websites within its network. Decentralized and secure, the domain names for these services are tied to public key infrastructure (PKI) but are challenged by their long and technical addresses. In response to this difficulty, in this thesis I introduce a novel and decentralized Tor-powered DNS system that provides unique and human-meaningful domain names to Tor hidden services.

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. This assumption is very similar to the assumption made when Tor clients create circuits: that all three nodes in a circuit are honest and less than all of them are semi-honest. Relays with stable and fast flags are have more consensus weight are thus more likely to be chosen relative to other nodes during circuit construction.

- 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 2.5.3) 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

SOLUTION

2.1 Overview

I propose a new distributed DNS, 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". [1] The system translates .tor domain names to .onion hidden service addresses using syntax adapted from the Clearnet DNS. In this chapter, first I describe the construction and contents of a distributed append-only ledger called a "page-chain", a data structure that holds DNS records and functionally resembles Namecoin's blockchain. Secondly, I describe how this page-chain can be used by the Tor network to power a publicly-verifiable distributed DNS system on top of existing Tor hidden service infrastructure. At a high level, EsgalDNS' master page-chain is maintained by a randomly-selected rotating set of high-capacity Tor nodes. Other Tor nodes may mirror the page-chain, distributing the load and responsibilities across the network. The system supports a variety of command and control operations including Create, Domain Query, Onion Query, Modify, Move, Renew, and Delete.

2.2 Cryptographic Primitives

EsgalDNS makes use of cryptographic hash algorithms, digital signatures, proof-of-work, and a pseudorandom number generator. We now describe our choices of fundamental cryptographic algorithms.

- 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. SHA-384 is

included in the National Security Agency's Suite B Cryptography for protecting up to Top Secret information.

- 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. 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. [2] 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.

2.3 Data Structures

EsgalDNS uses five main data structures: records, snapshots, pages, AVL trees, and a hashtable bitset. Here we describe these structures and how an individual machine can use them. The design assumes that the machine has access to Tor's consensus documents and can verify them; here we use the consensus documents as both a source of network and cryptographic information and as an agreed-upon time-based one-time password. In other words, the machine holding these structures can be assumed to be a Tor client.

2.3.1 Record

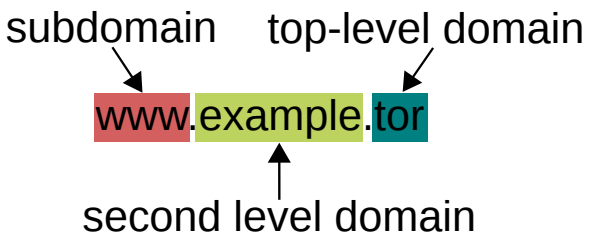

Figure 2.1: A sample domain name: a sequence of labels separated by delimiters. Here we use the .tor TLD and resolve the "www" third-level domain the same as the second-level domain by default.

A record is a simple data structure that represents a single DNS operation. Every record contains a public signing key, is self-signed, and contains a full list of domain names claimed by the issuer. Anyone may claim any second-level domain name that is not currently in use. There are five different types of records, each representing a corresponding control operation: Create, Modify, Move, Renew, and Delete.

As a distributed system has no central authorities from which one can purchase domain names, it is necessary to implement a system that introduces a cost of ownership. This fulfils three main purposes:

1. Significantly reduces the threat of record flooding.

2. Introduces a cost of investment that encourages the use of domain names and the availability of their servers.

3. Increases the difficulty of domain squatting, a denial-of-service where one claims one or more second-level domains for the purpose of making them unavailable to others. This attack is particularly difficult to resolve as records do not contain any personally-identifying information.

We introduce a proof-of-work system based on scrypt into all records. Proof-of-work schemes are noteworthy for their asymmetry: they require the issuer to find an answer to a computational problem that is feasible but moderately hard, but once found the solution is easily verified by any recipient. Proof-of-work is used in Hashcash, Namecoin, and in other cryptocurrencies.

Create

A Create record consists of nine components: *type*, *nameList*, *contact*, *timestamp*, *consensusHash*, *nonce*, *pow*, *recordSig*, and *pubHSKey*. Fields that are optional are blank unless specified, and all fields are encoded in base64, except for *nameList* and *timestamp*, which are encoded in standard UTF-8. These are defined in Table 2.3.1.

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 and their destinations. which can be domains with either the .tor or .onion TLDs. Domain names can be up to 16 levels deep and each name can be up to 32 characters long, though the full domain name cannot exceed 128 characters in length. This field must also contain second-level domain names, which must match any subdomains (names at level < 2) claimed by the issuer. In this way, records can be referenced by their unique master second-level domain names and issuers cannot claim ownership of domain names that they do not own.
contact	No	The final 16, 24, or 32 characters in the issuer's PGP key fingerprint, if he has a key chooses to disclose it. If the fingerprint is listed, clients may query a public keyserver for this fingerprint, obtain the PGP public key, and contact the owner over encrypted email.
timestamp	Yes	The UNIX timestamp of when the issuer created the registration and began the proof-of-work to validate it. This timestamp cannot be more than 48 hours old nor in the future relative to the recipient's clock.
consensusHash	Yes	The SHA-384 hash of a consensus document published at 00:00 GMT no more than 48 hours before the record was received by the recipient.
nonce	Yes	Four bytes that serve as a source of randomness for the proof-of-work.
pow	Yes	16 bytes that store the result of the proof-of-work.

recordSig	Yes	The digital signature of all preceding fields, signed using the issuer's private key.
pubHSKey	Yes	The issuer's public key in PKCS.1 DER encoding. When the SHA-1 hash of the decoded key is converted to base58 and truncated to 16 characters the resulting hidden service address must match at least one *nameList* destination.

Table 2.1: A Create record, which contains fields common to all records. Every record is self-signed and must have verifiable proof-of-work before it is considered valid.

Issuers must complete the proof-of-work before transmitting the record to the recipient. Let the variable *central* consist of all fields except *recordSig* and *pow*. The issuer must then find a *nonce* such that the SHA-384 of *central*, *pow*, and *recordSig* is $\leq 2^{\text{difficulty} * \text{count}}$, where *difficulty* specifies the order of magnitude of the work that must be done and *count* is the number of second-level domain names claimed. For each *nonce*, *pow* and *recordSig* must be regenerated. When the proof-of-work is complete, the valid and complete record is represented in JSON format and is ready for transmission.

Modify

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

Move

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

Renew

Second-level domain names (and their associated domain names) expire every L days because (as explained below) the page-chain has a maximum length of L pages. Renew records must be reissued periodically at least every L days to ensure continued ownership of domain names. Renew records are identical to Create records, except that *type* is set to "Renew". No modifications to existing domain names can be made in Renew records, and the domain names contained within must already exist in the page-chain. Similar to the Modify and Move records, Renew records have a difficulty of $\frac{\text{difficulty}_{Create}}{4}$.

Delete

If a owner's private key is compromised or if they wish to relinquish ownership rights over all of their domain names, they can issue a Delete record. Aside from their *type* field set to "Delete", Delete records are identical in form to Create records but have the opposite effect: all second-level domains contained within the record are purged from local caches and made available for others to claim. There is no difficulty associated with Delete records, so they can be issued instantly.

2.3.2 Snapshot

A *snapshot* is JSON-encoded textual database designed as a short-term and volatile cache. When two or more machines are maintaining a common page-chain, snapshots are

used as a buffer that is flushed to the other parties every Δs minutes. Individually, they are flushed to a local page every Δs minutes. Snapshots contain four fields: *originTime*, *recentRecords*, *fingerprint*, and *snapshotSig*.

originTime

Unix time when the snapshot was first created. This must be less than Δs minutes ago and not in the future relative to a recipient's clock.

recentRecords

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

fingerprint

The hash of the public key of the machine maintaining this snapshot.

snapshotSig

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

2.3.3 Page

A *page* is long-term JSON-encoded textual database used for archival of records. Each page contains a link to a previous page, forming an append-only public ledger known as a page-chain. In section 2.4 I describe how an individual machine can maintain its own page-chain and in section 2.5 I detail how nodes in the Tor network can maintain a common page-chain. In any fully-connected network where all participants know everyone's public keys, the page-chain becomes publicly confirmable and can be used for distributed DNS.

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

prevHash

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

recordList

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

consensusDocHash

The SHA-384 of *cd*.

fingerprint

The hash of the public key of the machine maintaining this page.

pageSig

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

2.3.4 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.

2.3.5 Hashtable Bitset

A hashtable bitset is a special and highly compact adaptation of a traditional hashtable. Unlike its AVL tree counterpart, its purpose to prove the non-existence of a record. As an extension of an ordinary hashtable, the hashtable bitset maps keys to buckets, but here we are interested in only tracking the existence of keys and have no need to store the keys themselves. Therefore each bucket in the structure is represented as a bit, creating a compact bitset of $\mathcal{O}(n)$ size. 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 second-level domains that map to that bucket should be added to an array list. Once the bitset is fully constructed, the array is sorted alphabetically and the resulted converted into the leaves of a Merkle tree.

Let Alice and a trusted authority Faythe maintain a common page-chain, and let Alice be a resolver for a third-party Bob. In the event that Bob asks Alice to resolve a domain name to a hidden service and Alice claims that the domain name does not exist, Bob must be

able to verify the non-existence of that record. Demonstrating non-existence is a challenge often overlooked in DNS: even if existing records can be authenticated by Bob, Alice may still lie and claim a false negative on the existence of a domain name. Bob may choose to query Faythe to confirm non-existence, but this introduces additional load onto Faythe. Bob cannot easily determine the accuracy of Alice's claim without downloading all of the records and confirming for himself, but this is impractical in most environments.

The hashtable bitset efficiently resolves this issue. Faythe periodically generates a timestamped hashtable bitset for all existing domain names, digitally signs it, and publishes the result to Alice. As described in section 2.4.3, Alice then includes this bitset along with any non-existence responses sent back to Bob. Since Bob knows and trusts Faythe, he can verify Alice's claim by verifying the authenticity of the bitset and confirming that the domain he requested does not appear in the hashtable or in the Merkle tree in $\mathcal{O}(1)$ time on average. The hashtable also serves as an optimization to Alice: she can check the bitset for Bob's request in $\mathcal{O}(1)$ time, bypassing the $\mathcal{O}(logn)$ lookup if the second-level domain name does not exist.

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

2.3.6 Trie

A trie, also known as a digital tree, is used for efficiently resolving Onion Queries (section 2.4.4). Each node in the trie corresponds to a base58 digit of the .onion hidden service address. These addresses are currently defined at 16 characters, so the trie has a maximum depth of 16 and up to 58 branches per node.

2.4 Algorithms

2.4.1 Initialization

This following description assumes that Alice is maintaining a local page-chain in synchronization with a Faythe, a trusted second party. The functionality described here is extended to distributed environments in 2.5 – Application Within Tor.

Alice creates an initial EsgalDNS database by the following procedure. Let i be the current day, and let Δi be the lifetime in days of a individual page.

1. Download a copy of the consensus document *cd* published at at 00:00 GMT on day $\left\lfloor \frac{i}{\Delta i} \right\rfloor$.

2. Create an empty AVL tree, an initial hashtable bitset, an empty trie, a blank snapshot, and an empty page.

3. Sets the snapshot's *originTime* field to the current Unix time, *recentRecords* to an empty array, and *fingerprint* to the hash of her public key.

4. Digitally sign *originTime* + *recentRecords* + *fingerprint* and save the signature in *snapshotSig*.

5. Set the page's *prevHash* field to zeros, *recordList* to an empty array, *fingerprint* to the hash of her public key, and *consensusDocHash* to the SHA-384 of *cd*.

6. Digitally sign *prevHash* + *recordList* + *consensusDocHash* + *fingerprint* and save the signature in *pageSig*.

2.4.2 Page-chain Maintenance

Alice then listens for new records from herself or from third-parties. Let p_i be Alice's current page and s_x the current snapshot.

For each received record, Alice

1. Rejects the record if the fields are invalid, the signature does not validate, or the proof-of-work cannot be confirmed.

2. Checks for the second-level domain name in s_x, p_i, the hashtable bitset, and then the AVL tree.

3. If Alice received a Create record, rejects the record if the domain was found.

4. If a Modify, Move, Renew or Delete record was received, rejects the record if the domain was not found.

5. Adds the record into s_x's *recentRecords* and regenerates *snapshotSig*.

Every Δs minutes, Alice

1. Generates a new snapshot, s_{x+1}.

2. Sets its *originTime* to the current time, creates *snapshotSig*, and sets s_{x+1} to be the currently active snapshot for collecting new records.

3. Sends s_x to Faythe and accepts Faythe's snapshot if one is offered.

4. Merge all records in *recentRecords* from s_x and Faythe's snapshot into the *recordList* field of p_i.

5. Regenerates *pageSig*.

6. Increments x.

Every Δi days, Alice

1. Creates a new page p_{i+1} and sets its *prevHash* to SHA-384(*prevHash* + *recordList* + *consensusDocHash*) from p_i.

2. Sets p_{i+1}'s *consensusDocHash* to the SHA-384 of the consensus document at 00:00 GMT.

3. Deletes page p_{i-L} if it exists.

4. Regenerates her local AVL tree and hashtable bitset from records in the page-chain using the pages in reverse chronological order. The leaves of the AVL tree at each second-level domain name point to the latest record containing that domain. Deletion records mark that domain as available and invalidate any Create, Modify, Move, or Renew that contain that domain earlier in the page-chain.

5. Regenerates the trie with the .onion addresses in the page-chain such that each leaf in the trie is mapped to the record that immediately resolves it.

6. Asks Faythe for an updated signature on her hashtable bitset, which should verify against Alice's bitset.

7. Increments i.

2.4.3 Domain Query

Alice can optionally act as a resolver for other parties. Let Bob be a client and Faythe a third-party trusted by both Alice and Bob. Assume that Bob does not trust Alice, Bob has Faythe's public key, and that Bob has obtained a copy of the Tor consensus document published on day $\lfloor \frac{i}{\Delta i} \rfloor$ at 00:00 GMT. Faythe has divided her bitset into Q sections, digitally signed each section, digitally signed the root hash of the Merkle tree, and send the signatures to Alice.

Bob enters "sub.example.tor" into his Tor Browser. As this TLD is not used by the Clearnet DNS, his client software directs the request to Alice. Bob must recursively resolve the .tor domain name into a .onion address and it is more efficient if Alice returns back to Bob all the necessary information that he needs to perform this resolution. Along with the domain name, Bob also sends to Alice one of the two verification levels, each providing progressively more verification that the record Bob receives is authentic, unique, and trustworthy. The default verification level is 0 for performance reasons.

At verification level 0, Alice first checks the hashtable bitset to confirm that the record exists. If it does, Alice then queries the AVL tree to find the latest record that contains the requested domain name. Alice resolves the requested domain and repeats this lookup

up to eight times if the resulting destination uses the .tor TLD. Once a .onion TLD destination is encountered, Alice returns to Bob all records containing the intermediate and final destinations. If a lookup fails, Alice returns to Bob records containing any intermediary destinations, and the Faythe-signed section of the bitset. If the hash maps to a "1" bucket, Alice also returns the Faythe-signed root of the Merkle collision tree, the leaves of the branch containing neighbouring second-level domain names that alphabetically span the failed domain, and all other hashes at the top level of the branch in the Merkle tree.

Bob receives from Alice the data structures containing the domain name resolutions. Bob confirms the resolution path and confirms the validity, signature, and proof-of-work of each record in turn. Finally, Bob looks up the .onion hidden service in the traditional manner. If the resolution was unsuccessful, Bob also verifies the signed section of the bitset. If the domain maps to a "0", Bob's client software returns that the DNS lookup failed, otherwise Bob confirms that no such second-level domain name exists in the Merkle tree branch returned by Alice. Since the leaves of the branch alphabetically span the requested domain and would otherwise contain it if it existed, Bob knows that it does not exist. Finally, Bob hashes the leaves to reconstructs the branch, and constructs the rest of the tree by combining the root of that branch with the other hashes returned by Alice at that level, and verifies the root hash against Faythe's signature. If this validates, Bob's software also informs the Tor Browser that DNS lookup failed. The resolution can also fail if the service has not published a recent hidden service descriptor to Tor's distributed hash table. End-to-end verification (relative to the authenticity of the hidden service itself) 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, in addition to returning all record and supplementary data structures in level 0, Alice also returns the page that contained each record and the *pageSig* field from Faythe's page. Since Alice and Faythe are maintaining a common page-chain, both Alice and Faythe will be signing the same data. Bob verifies the authenticity of the page against both Alice and Faythe's public keys and proceeds with his steps in level 0.

2.4.4 Onion Query

Bob may also issue a reverse-hostname lookup to Alice to find second-level domains that directly resolve to a given .onion hidden service address. This request is an Onion Query. Unlike on the Clearnet (under RFCs 1033 and 1912) not every .onion name has a corresponding domain name, so these queries may also fail. When Bob sends a Onion Query to Alice, Alice finds in her trie the record that contains a second-level .tor domain name that immediately maps to that .onion, and returns the domain name. Bob can optionally perform additional verification by issuing a Level 1 Domain Query on that domain name which should resolve to the original requested .onion hidden service address, a forward-confirmed reverse DNS lookup.

2.4.5 Synchronization

Rather than continue to query Alice, Bob may wish to become his own resolver. Bob may also become a resolver for others that trust Faythe. The procedure to clone Alice's databases is simple:

1. Bob downloads Alice's $\min(i, L)$ most recent pages in her page-chain.

2. Bob obtains the consensus documents published every Δi days between days $i - \min(i, L)$ and i at 00:00 GMT. Bob may download these from Alice, but Bob may also download them from any other source. These documents may be compressed beforehand: : very high compression ratios can be achieved under 7zip.

3. Bob verifies the authenticity of the consensus documents and then verifies the signatures on each page in the page-chain.

4. Bob constructs his own AVL tree, hashtable bitset, and trie and confirms that Faythe's signatures on her bitset and Merkle collision tree root match his copy. If so, there has been no corruption and Bob has an authentic copy of the database.

Once the synchronization is complete, Bob can confirm the integrity, authenticity, and uniqueness of domain names.

2.5 Application Within Tor

When used inside a fully connected network such as Tor, individual nodes continue to hold their own page-chains and if they share snapshots with each other the network will, under ideal conditions, maintain a common page-chain and general database. This is possible if the Faythe character is represented by the whole or part of the rest of the network. Specifically, rather than flush s_x every Δs minutes to a single trusted source and check their *pageSig*s, the communication happens with many authorities. In this situation, distributed DNS can be achieved.

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.

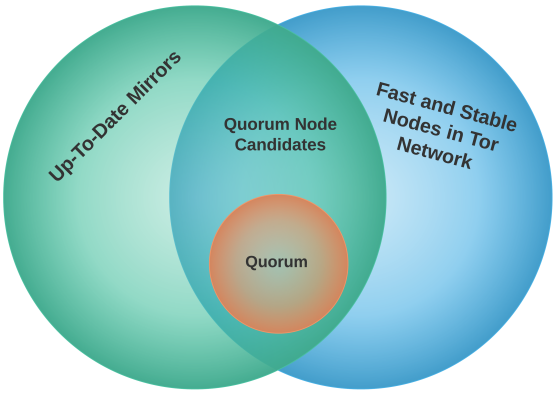

Figure 2.2: There are three sets of participants in the EsgalDNS network: *mirrors*, quorum node *candidates*, and *quorum* members. The set of *quorum* nodes is chosen from the pool of up-to-date *mirrors* who are reliable nodes within the Tor network.

2.5.1 Mirrors

Mirrors are Tor nodes that have performed a full synchronization (section 2.4.5) 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.

2.5.2 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 [3]: 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 2.3.4.
2. Encode SHA-384(*tree*) in Base64 and truncate to 8 bytes.
3. 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. One possible solution is to combine this hash publication with a Time-based One-time Password Algorithm (TOTP) at a 1 hour time interval. Although this would not thwart collusion or the publishing of the hash elsewhere, it would make the attack non-trivial.

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.

2.5.3 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:

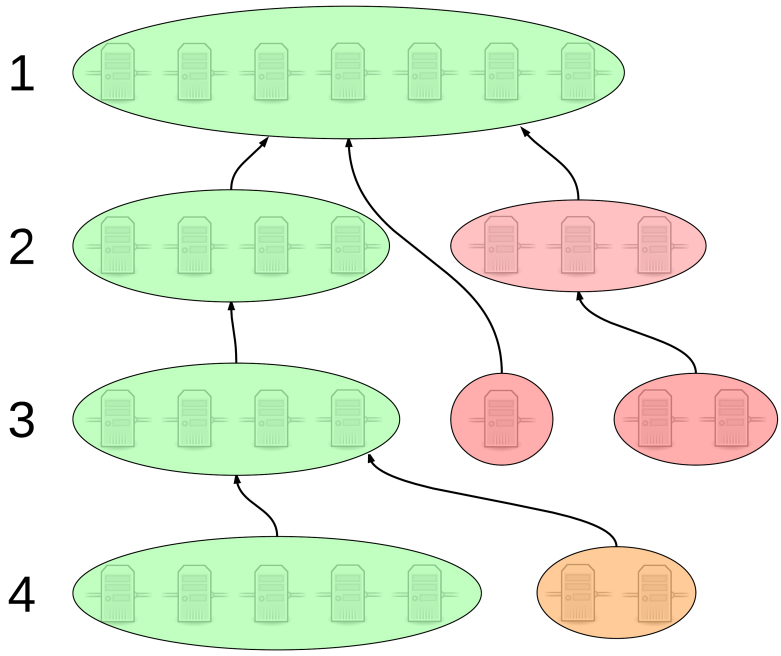

Figure 2.3: An example *page*-chain across four *quorums*. Each *page* contains a references to a previous *page*, forming an distributed scrolling data structure. *Quorums* 1 is semi-honest and maintains its uptime, and thus has identical *pages*. *Quorum* 2's largest cluster is likewise in agreement, 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.

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 pages and the *quorum* both change every Δi days, *quorum* nodes have an effective lifetime of Δi days before they are replaced by a new *quorum*.

2.5.4 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 $\left\lfloor \frac{i}{\Delta i} \right\rfloor$ 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. 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.

This process is illustrated in figure 2.4.

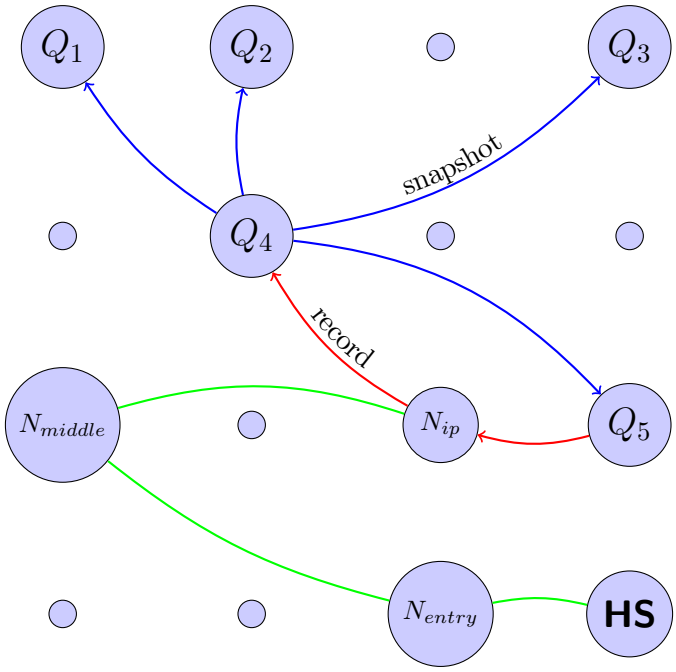

Figure 2.4: The hidden service operator uses his existing circuit (green) to inform *quorum* node Q_4 of the new record. Q_4 then distributes it via *snapshots* to all other *quorum* nodes. Each records it in their own *page* for long-term storage. The operator also confirms from a randomly-chosen *quorum* node Q_5 that the record has been received.

CHAPTER 3

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.

3.1 Security

3.1.1 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 [4]) 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.

3.1.2 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.

3.1.3 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.

3.2 Performance

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

3.2.1 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.

3.3 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 4

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] 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.

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

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

[4] 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.