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

ABSTRACT

EsgalDNS:

Tor-powered Distributed DNS

for Tor Hidden Services

by

Jesse Victors, Master of Science

Utah State University, 2015

Major Professor: Dr. Ming Li
Department: Computer Science

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 have usability challenges due to their long and technical addresses. In response to this difficulty, I propose and partially implement a decentralized DNS system inside the Tor network. The system provides a secure and verifiable mapping between human-meaningful names and traditional Tor hidden service addresses, is backwards- and forwards-compatible with Tor hidden service infrastructure, and preserves the anonymity of the hidden service and its operator.

(51 pages)

This work is dedicated to the developers and community behind the Tor Project and the Tails OS. These individuals work tirelessly to preserve the privacy and security of everyday citizens, journalists, activists, and others around the globe, providing the much-needed service of private conversations in a very public world.

CONTENTS

Page

ABSTRACT iii

LIST OF FIGURES vii

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND 2
2.1 Tor 2
2.2 Motivation 7

3 REQUIREMENTS 8
3.1 Assumptions and Threat Model 8
3.2 Design Principles 8

4 CHALLENGES 11
4.1 Zooko's Triangle 11
4.2 Communication 12
4.3 Fault Tolerance 12

5 EXISTING WORKS 13
5.1 Encoding Schemes 13
5.2 Clearnet DNS 15
5.3 Namecoin 16

6 SOLUTION 20
6.1 Overview 20
6.2 Components 20
6.3 Examples and Structural Induction 33
6.4 Fault Tolerance 37

7 ANALYSIS 38
7.1 Security 38
7.2 Performance 39

8 RESULTS 40
8.1 Implementation 40
8.2 Discussion 40

9 FUTURE WORK 41

10 CONCLUSION 42

REFERENCES 43

LIST OF FIGURES

Figure		Page
2.1	Anatomy of the construction of a Tor circuit.	4
2.2	A circuit through the Tor network.	4
2.3	A Tor circuit is changed periodically, creating a new user identity.	5
2.4	Alice uses the encrypted cookie to tell Bob to switch to *rp*.	6
2.5	Bidirectional communication between Alice and the hidden service.	6
4.1	Zooko's Triangle.	11
5.1	Three traditional Namecoin transactions.	17
5.2	A sample blockchain.	18
6.1	Sample page-chain on day 4. Quorum size is $M = 10$, the numbers represent quorum participants with the same page. Despite malicious collusion and misbehaving nodes on day 3, most of day 4's quorum choose the correct chain with $10 + 9 + 1 + 5 + 4 + 2 = 31$ total participants.	26
6.2	HS operator broadcasts a record though a circuit (green) to a quorum node. The record propagates through snapshots.	31
6.3	Sample empty initial page, $p_{1,1}$	33
6.4	Sample registration record from a hidden service. Note that the "sub.example.tor" $\rightarrow$ "example.tor" $\rightarrow$ "exampleruyw6wgve.onion" references can be resolved recursively.	34
6.5	Sample snapshot from c_1, containing one registration record r_{reg} from a hidden service.	35
6.6	c_1's page, containing a single registration record.	36

CHAPTER 1

INTRODUCTION

The Tor network is a second-generation onion routing system that aims to provide anonymity, privacy, and Internet censorship protection to its users. The Tor client software multiplexes all end-user TCP traffic through a series of relays on the Tor network, typically a carefully-constructed three-hop path known as a *circuit*. Each relay in the circuit has its own encryption layer, so traffic is encrypted multiple times and then is decrypted in an onion-like fashion as it travels through the Tor circuit. As each relay sees no more than one hop in the circuit, in theory neither an eavesdropper nor a compromised relay can link the connection's source, destination, and content. Tor remains one of the most popular and secure tools to use against network surveillance, traffic analysis, and information censorship.

While the majority of Tor's usage is for traditional access to the Internet, Tor's routing scheme also supports anonymous websites, hidden inside Tor. Unlike the Clearnet, Tor does not contain a traditional DNS system for its websites; instead, hidden services are identified by their public key and can be accessed through Tor circuits. A client and the hidden service can thus communicate anonymously.

CHAPTER 2

BACKGROUND

2.1 Tor

The Tor network is a third-generation onion routing system, originally designed by the U.S. Naval Research Laboratory for protecting sensitive government communication. Tor refers both to the client-side multiplexing software and to the worldwide volunteer-run network of over six thousand nodes. The Tor software provides an anonymity and privacy layer to end-users by relaying TCP traffic through a series of relays on the Tor network. Tor sees global widespread use and as of January 2015 has over 2.4 million daily users and passes an average of approximately 6000 MB per second through its network. [1] Tor's encryption and routing protocols are designed to make it very difficult for an adversary to correlate an end user to their traffic. Tor has been recognized by the NSA as the "the king of high secure, low latency Internet anonymity". [2]

2.1.1 Design

Tor routes encrypted TCP/IP user traffic through a worldwide volunteer-run network of over six thousand relays. Typically this route consists of a carefully-constructed three-hop path known as a *circuit*, which changes over time. These nodes in the circuit are commonly referred to as *guard node*, *middle relay*, and the *exit node*, respectively. Only the first node is exposed to the origin of TCP traffic into Tor, and only the exit node can see the destination of traffic out of Tor. The middle router, which passes encrypted traffic between the two, is unaware of either. The client negotiates a separate TLS connection with each node at a time, and traffic through the circuit is decrypted one layer at a time. As such, each node is only aware of the machines it talks to, and only the client knows the

identity of all three nodes used in its circuit, making traffic correlation much more difficult difficult compared to a VPN, proxy, or a direct TLS connection.

The Tor network is maintained by nine authority nodes, who each vote on the status of nodes and together hourly publish a digitally signed consensus document containing IPs, ports, public keys, latest status, and capabilities of all nodes in the network. The document is then redistributed by other Tor nodes to clients, enabling access to the network. The document also allows clients to authenticate Tor nodes when constructing circuits, as well as allowing Tor nodes to authenticate one another. Since all parties have prior knowledge of the public keys of the authority nodes, the consensus document cannot be forged or modified without disrupting the digital signature. [3]

2.1.2 Routing

In traditional Internet connections, the client communicates directly with the server. TLS encryption cannot hide IP and TCP headers, which must be exposed to allow routing. Eeavesdroppers can track end-users by monitoring these headers, easily correlating clients to their activities. Tor combats this by routing end user traffic through a randomized circuit through the network of relays. The client software first queries an authority node or a known relay for the latest consensus document. Next, the Tor client chooses three unique and geographically diverse nodes to use. It then builds and extends the circuit one node at at time, negotiating respective TLS connections with each node in turn. No single relay knows the complete path, and each relay can only decrypt its layer of decryption. In this way, data is encrypted multiple times and then is decrypted in an onion-like fashion as it passes through the circuit.

Todo: the following paragraph is not a complete description of Tor's crypto. I don't describe the NTor routing protocol.

The client first establishes a TLS connection with the first relay, R_1, using the relay's public key. The client then performs an ECDHE key exchange to negotiate K_1 which is then used to generate two symmetric session keys: a forward key $K_{1,F}$ and a backwards key $K_{1,B}$. $K_{1,F}$ is used to encrypt all communication from the client to R_1 and $K_{1,B}$ is used

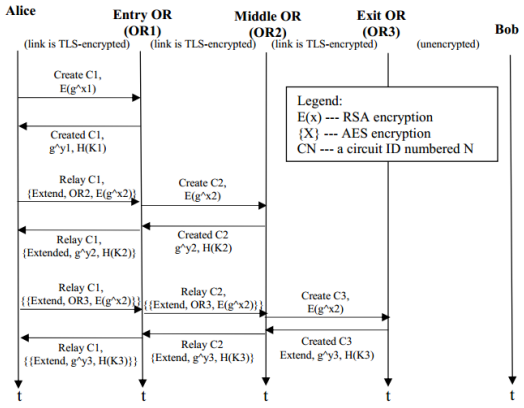

Figure 2.1: Anatomy of the construction of a Tor circuit.

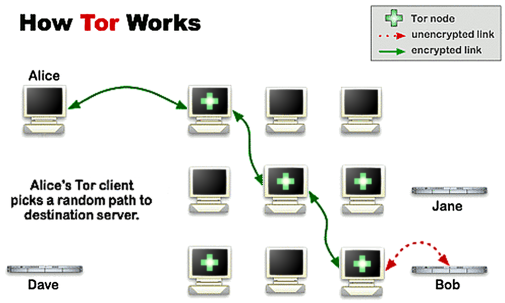

Figure 2.2: A circuit through the Tor network.

for all replies from R_1 to the client. These keys are used conjunction with the symmetric cipher suite negotiated during the TLS handshake, thus forming an encrypted tunnel with perfect forward secrecy. Once this one-hop circuit has been created, the client then sends R_1 the RELAY_EXTEND command, the address of R_2, and the client's half of the Diffie-Hellman-Merkle protocol using $K_{1,F}$. R_1 performs a TLS handshake with R_2 and uses R_2's public key to send this half of the handshake to R_2, who replies with his second half of the handshake and a hash of K_2. R_1 then forwards this to the client under $R_{1,B}$ with the RELAY_EXTENDED command to notify the client. The client generates $K_{1,F}$ and $K_{1,B}$ from K_2, and repeats the process for R_3, [4] as shown in Figure 3. The TCP/IP connections remain open, so the returned information travels back up the circuit to the end user.

Following the complete establishment of a circuit, the Tor client software then offers a Secure Sockets (SOCKS) interface on localhost which multiplexes any TCP traffic through Tor. At the application layer, this data is packed and padded into equally-sized Tor *cells*, transmission units of 512 bytes. As each relay sees no more than one hop in the circuit, in theory neither an eavesdropper nor a compromised relay can link the connection's source, destination, and content. Tor further obfuscates user traffic by changing the circuit path every ten minutes, [5] as shown in Figure 4. A new circuit can also be requested manually by the user.

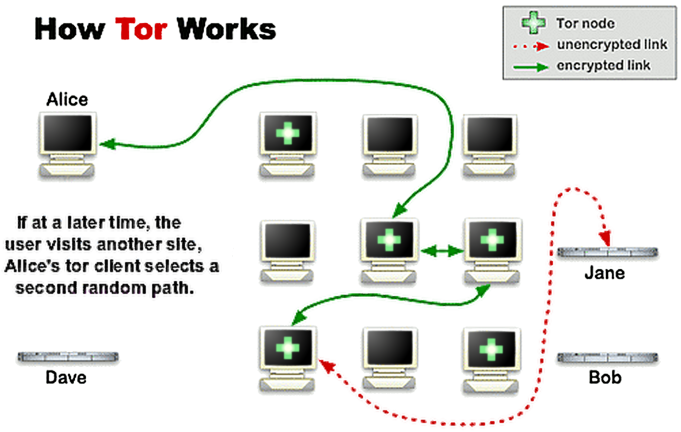

Figure 2.3: A Tor circuit is changed periodically, creating a new user identity.

Tor users typically use the Tor Browser, a custom build of Mozilla Firefox with a focus on security and privacy. The TBB anonymizes and provides privacy to the user in many ways. These include blocking all web scripts not explicitly whitelisted, forcing all traffic including DNS requests through the Tor SOCKS port, mimicking Firefox in Windows both with a user agent (regardless of the native platform) and SSL cipher suites, and reducing Javascript timer precision to avoid identification through clock skew. Furthermore, the TBB includes the Electronic Frontier Foundation's HTTPS Everywhere extension, which uses regular expressions to rewrite HTTP web requests into HTTPS for domains that are known to support HTTPS. If this is the case, an HTTPS connection will be established with the web server. If this happens, end-to-end encryption is complete and an outsider near the user would be faced with up to four layers of TLS encryption: $K_{1,F}(K_{2,F}(K_{3,F}(K_{server}(\text{client request}))))$ and likewise $K_{1,B}(K_{2,B}(K_{3,B}(K_{server}(\text{server reply}))))$ for the returning traffic, making traffic analysis very difficult.

2.1.3 Hidden Services

Although Tor's primary and most popular use is for secure access to the traditional Internet, since 2004 Tor also supports anonymous services, such as websites, marketplaces, or chatrooms. These are a part of the Dark Web and cannot be normally accessed outside the context of Tor. In contrast to Tor-anonymized web requests where the client is anonymous

but the server is known, Tor hidden services provide bidirectional anonymity where both parties remain anonymous and never directly communicate with one another. This allows for a greater range of communication capabilities. [6]

Tor hidden services are known only by their public RSA key. Tor does not contain a DNS system for its websites; instead the domain names of hidden services are an 80-bit truncated SHA-1 hash of its public key, postpended by the .onion top-level domain (TLD). Once the hidden service is contacted and its public key obtained, this key can be checked against the requested domain to verify the authenticity of the service server. This process is analogous to SSL certificates in the clearnet, however Tor's authenticity check leaks no identifiable information about the anonymous server. If a client obtains the hash domain name of the hidden service through a backchannel and enters it into the Tor Browser, the hidden service lookup begins.

Preceding any client communication, the hidden server, Bob, first builds Tor circuits to several random relays and enables them to act as *introduction points* by giving them its public key, B_K. The server then uploads its public key and the fingerprint identity of these nodes to a distributed hashtable inside the Tor network, signing the result. When a client, Alice, requests contact with Bob, Alice's Tor software queries this hashtable, obtains B_K and Bob's introduction points, and builds a Tor circuit to one of them, ip_1. Simultaneously, the client also builds a circuit to another relay, rp, which she enables as a rendezvous point by telling it a one-time secret, sec.

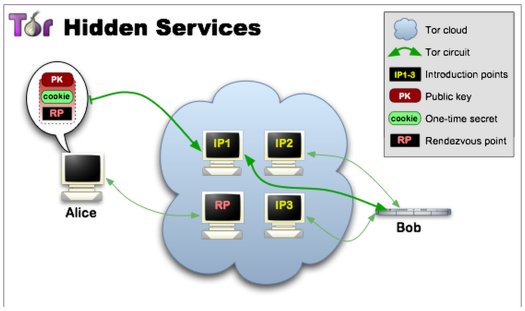

Figure 2.4: Alice uses the encrypted cookie to tell Bob to switch to rp.

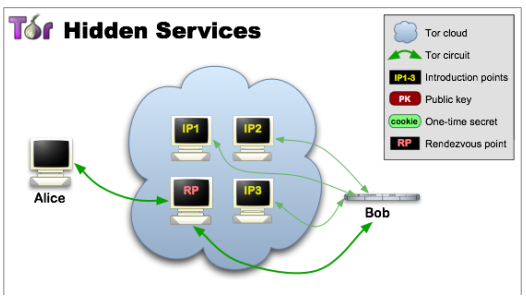

Figure 2.5: Bidirectional communication between Alice and the hidden service.

She then sends to ip_1 a cookie encrypted with B_K, containing rp and sec. Bob decrypts this message, builds a circuit to rp, and tells it sec, enabling Alice and Bob to communicate. Their communication travels through six Tor nodes: three established by Alice and three by Bob, so both parties remain anonymous. From there traditional HTTP, FTP, SSH, or other protocols can be multiplexed over this new channel.

2.2 Motivation

The usability of hidden services is severely challenged by their non-intuitive 16-character base58-encoded domain names. To choose several prominent examples, 3g2upl4pq6kufc4m.onion is the address for the DuckDuckGo hidden service, 33y6fjyhs3phzfjj.onion is the Guardian's SecureDrop service for anonymous document submission, and blockchainbdgpzk.onion is the anonymized edition of blockchain.info. It is rarely clear what service a hidden server is providing by its domain name alone without relying on third-party directories for the correlation, directories which must be updated and reliably maintained constantly. These must be then distributed through backchannels such as /r/onions, the-hidden-wiki.com, or through a hidden service that is known in advance. It is a frequent topic of conversation inside Tor communities. It is clear that this problem limits the usability and popularity of Tor hidden services. Although there have been some workarounds that are partially successful, the issue remains unresolved. It is for these reaons that I propose EsgalDNS as a full solution.

CHAPTER 3

REQUIREMENTS

3.1 Assumptions and Threat Model

One of the primary assumptions that I make in this system is that not all Tor nodes can be trusted. Some of them may be run by malicious operators, curious researchers, or experimenting developers. They may be wiretapped, the Tor software modified and recompiled, or they may otherwise behave in an abnormal fashion. I assume that adversaries have control over some of the Tor network, they have access to large amounts of computational and financial resources, and that they have access to portions of Internet traffic, including portions of Tor traffic. I also assume that they do not have global and total Internet monitoring capabilities and I make no attempt to defend against such an attacker, although it should be noted this is an assumption also made by Tor. I work under the belief that attackers are not capable of cryptographically breaking properly-implemented TLS connections and their modern components, particularly 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. Lastly, I assume that adversaries monitor and may attempt to modify the DNS record databases, but I assume that at least 50 percent of the Tor network is trustworthy and behave normally in accordance with Tor and EsgalDNS specifications.

3.2 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, including over the wire.

2. Lookups must be anonymous or at least privacy-enhanced; it should not be trivial to determine what hidden services a client is interested in.

3. Registrations must be publicly confirmable; clients 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 hashes.

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, even 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 4

CHALLENGES

4.1 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: [7]

- 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.

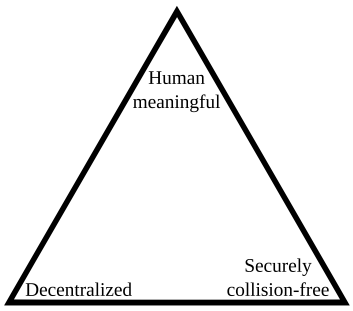

Figure 4.1: Zooko's Triangle.

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. [8]

4.2 Communication

4.3 Fault Tolerance

CHAPTER 5

EXISTING WORKS

Several workarounds exist that attempt to alleviate the issue, with one used in practice, but none are fully successful. Other DNS systems already exist, including some that are distributed, but they are either not applicable, or their design makes it extremely challenging to integration into the Tor environment. Two primary problems occur when attempting to use existing DNS systems: being able to prove the correlation of ownership between the DNS name and the hidden service, and the leakage of information that may compromise the anonymity of the hidden service or its operator. As hidden services are only known by their public key and introduction points, it is not easy to use only the hidden service descriptor to prove ownership, and there are privacy and security issues with requiring any more information. Due to these problems, no existing works have been yet integrated into the Tor environment, and the one workaround that is used in practice remains only partially successful.

5.1 Encoding Schemes

In an attempt to address the readability and the memorability of hidden service domain names, two changes to the encoding of domain names have been proposed, with one used in practice.

Shallot, originally created by an anonymous developer sometime between 2004 and 2010, is an application which uses OpenSSL to generate many RSA hidden service keys in an attempt to find one that has a desirable hash, such as one that begins with a meaningful noun. [9] By brute-forcing the domain key-space, Shallot will eventually find a domain that is meaningful and partially memorable to the hidden service operator. Shallot was used by Blockchain.info (blockchainbdgpzk.onion), by Facebook (facebookcorewwwi.onion), and by

many other hidden services in an attempt to make their domain name appear less random. However, Shallot only partially successful because the size of the domain key-space is too large to be fully brute-forced in any reasonable length of time. For example, on a 1.5 GHz processor, Shallot is expected to take approximately 25 days to find a key whose hash contains 8 custom letters out of the 16 total. [9] Tor Proposal 224 makes this solution even worse as it suggests 32-byte domain names which embed an entire ECDSA hidden service key in base32. [10] Although prefixing the domain name with a meaningful word helps identify a hidden service at a glance, it does nothing to alleviate the logistic problems of entering a hidden service domain name, including the remaining random characters, manually into the Tor Browser.

A different encoding scheme was proposed in 2011 by Simon Nicolussi, who suggested that encoding the key hash as a series of words, using a dictionary known in advance by all parties. The list of words would then represent the domain name, rather than base58. While this scheme would improve the memorability of hidden services, the words cannot all be chosen by the hidden service operator, and brute-forcing with Shallot would again only be partially successful due to the large key-space. Therefore this solution could be used to generate some words that relate to the hidden service in a meaningful way, but this scheme is only a partial solution. [6]

These schemes do not change the underlying hidden service protocol, they just attempt to increase the readability of the domain names in Tor Browser. Compared against our original requirements, they meet the anonymity for registration and lookups due to Tor circuits, the confirmability because the domain is the hash of the public key, the uniqueness of domain names due to the collision-free property of SHA-1, the distributed requirement by way of the distributed hashtable stored throughout the Tor network, and resistance to malicious modifications because of the strong association between domain names and the hidden service key. However, it fails to meet the simplicity requirement because although hidden service domain names are entered in the traditional manner into the Tor Browser, the domain names are not entirely human-meaningful nor memorable. The domain names con-

tinue to suffer from usability problems, only partially alleviated by these encoding schemes. These workarounds do not introduce any new DNS systems or change the existing Tor hidden service protocol in any way that allows for customized domain names, but these partial solutions are worthy of mention nevertheless.

5.2 Clearnet DNS

The Internet Domain Name Service (DNS) was originally designed in 1983 as a hierarchical naming system to link domain names to Internet Protocol (IP) addresses and translate one to the other. IP addresses specify the location of a computer or device on a network and domain names identify that resource. Domain names therefore operate as an abstraction layer, allowing servers to be moved to a different IP address without loss of functionality. The names consist of a top-level domain (TLD) that is prefixed by a sequence of labels, delimited by dotes. The labels divide the DNS system into zones of responsibility, where each label can be unique within that zone, but not necessarily across different zones. Each label can consist of up to 63 characters and the domain names can be up to 253 characters in total length. In contrast to IP addresses, domain names are human-meaningful and easily memorized, so DNS is a crucial component to the usability of the Internet.

The Clearnet DNS system suffers from several significant shortcomings that make it inappropriate for use by Tor hidden services. First, responses to DNS queries are not authenticated by digital signatures, so spoofing by a MITM attack is possible. Secondly, queries and responses are not encrypted, so it is easy for anyone wiretapping port 53 to correlate end-users to their activities, even if the communication to those websites is protected by TLS/HTTPS. Third, it suffers from DNS cache poisoning, in which an attacker pretends to be an authoritative server and sends incorrect or malicious records to local DNS resolvers. Finally, owning a TLD specifically for Tor hidden services (such as .tor) is prohibitively expensive. While DNS looks traditionally go through the Tor circuit and are resolved by Tor exit nodes, the shortcomings of the Clearnet DNS system make it unsuitable for our purposes.

The traditional Clearnet DNS system meets only a few of our requirements; registrations require a significant of identifiable and geographic information and thus are far from anonymous, lookups occur without encryption or signatures and are therefore neither anonymous nor privacy-enhanced, registrations are by default not publically confirmable but can be made so through use of an expensive SSL certificate from a central authority, the system is zone-based but is managed by centralized organizations and is therefore only partially distributed, and while it is very simple to use, extremely popular, and backwards compatible with TCP/IP, a Tor-specific TLD does not exist and falsifications of registrations is possible in a number of different ways. For its many security issues we therefore dismiss it as a possible solution.

5.3 Namecoin

Namecoin (NMC) is a decentralized peer-to-peer information registration and transfer system, developed by Vincent Durham in early 2011. It was the first fork of Bitcoin and as such inherits most of Bitcoin's design and capabilities. Like Bitcoin, a digital cryptocurrency created by pseudonymous developer Satoshi Nakamoto in 2008, Namecoin holds name records and transactions in a public ledger known as a blockchain. Users may hold Namecoins, and may prove ownership and authorize transactions with their private secp256k1 ECDSA key. Each transaction is a transfer of ownership of a certain amount of NMC from one public key to another, and as all transactions are recorded into the blockchain by the Namecoin network, all Namecoins can be traced backwards to the point of origin. Purchasing a name-value pair, such as a domain name, consumes 0.01 Namecoins, thus adding value and some expense to names in the system. Durham added two rules to Namecoin not present in Bitcoin: names in the blockchain expire after 36,000 blocks (about every 250 days) unless renewed by the owner and no two unexpired names can be identical. Namecoin domain names use the .bit TLD, which is not in use by ICANN.

The blockchain data structure is Nakamoto's novel answer to the problem of ensuring agreement of critical data across all involved parties. This prevents the double-spending of Bitcoins, or in Namecoin's case the prevention of duplicate names. Starting from an

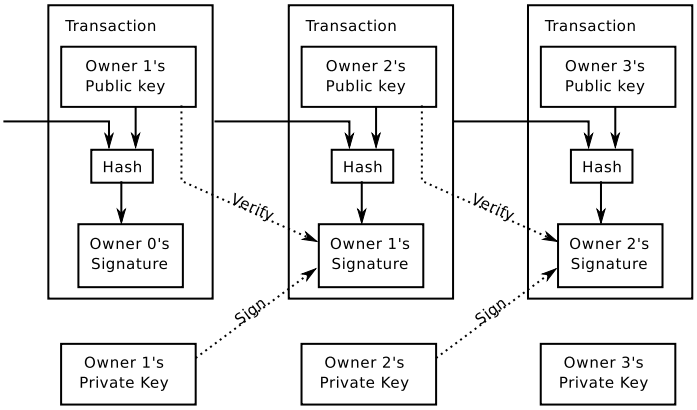

Figure 5.1: Three traditional Namecoin transactions.

initial genesis block, all blocks of data link to one another by referencing the hash of a previous block. Blocks are generated and appended to the chain at a relatively fixed rate by a process known as mining. The mining process is the solving of a proof-of-work (PoW) problem in a scheme similar to Adam Back's Hashcash: find a nonce that when passed through two rounds of SHA256 (SHA256^2) produces a value less than or equal to a target T. This requires a party to perform on average $\frac{1}{Pr[H \leq T]} = \frac{2^{256}}{T}$ amount of computations, but it is easy to verify afterwards that $\text{SHA256}^2(\text{msg}||n) \leq T$. Once the PoW is complete, the block (containing the back-reference hash, a list of transactions, and the PoW variables) is broadcasting to rest of the network, thus forming an append-only chain whose validity everyone can confirm. As each block is generated, the miner receives fresh Namecoins, thus introducing newly-minted Namecoins into the system at a fixed rate. Blocks cannot be modified retrospectively without requiring regeneration of the PoW of that block and all subsequent blocks, so the PoW locks blocks in the chain together. Nodes in the Namecoin network collectively agree to use the blockchain with the highest accumulation of computational effort, so an adversary seeking to modify the structure would need to recompute the proof-of-work for all previous blocks as well as out-perform the network, which is currently considered infeasible. [11]

Each block in the blockchain consists of a header and a payload. The header contains a hash of the previous block's header, the root hash of the Merkle tree built from the

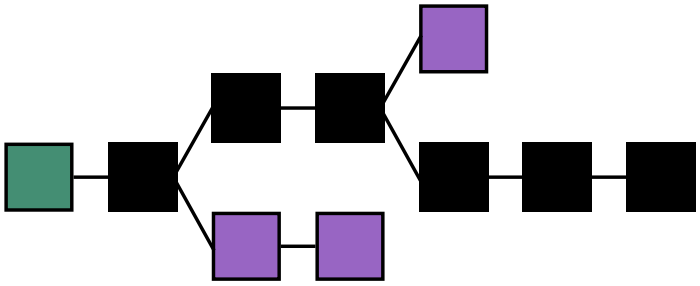

Figure 5.2: A sample blockchain.

transactions in this block, a timestamp, a target T, and a nonce. The block payload consists of a list of transactions. The root node of the Merkle tree ensures the integrity of the transaction vector: verifying that a given transaction is contained in the tree takes $log(n)$ hashes, and a Merkle tree can be built $n * log(n)$ time, ensuring that all transactions are accounted for. The hash of the previous block in the header ensures that blocks are ordered chronologically, and the Merkle root hash ensures that the transactions contained in each block are order chronologically as well. The target T changes every 2016 blocks in response to the speed at which the proof-of-work is solved such that Bitcoin miners take two weeks to generate 2016 blocks, or one block every 10 minutes. The change in target ensures that the difficulty of the proof-of-work remains relatively constant as processing capabilities increase according to Moore's Law. In the event that multiple nodes solve the proof-of-work and generate a new block simultaneously, the forked block becomes orphaned, the transactions recycled, and the network converges to follow the blockchain with the longest path from the genesis node, thus the one with the most amount of PoW behind it. [11]

Namecoin is particularly noteworthy in that it was the first systems to prove Zooko's Triangle false in a practical sense. The blockchain public ledger is held by every node in the distributed network, and human-meaningful names can be owned and placed inside it. The rules of the Namecoin network tell all participants to ensure that there are no name duplicates, and anyone holding the blockchain can verify this property. Names cannot be inserted retroactively due to the PoW, so these properties are ensured. Thus, Namecoin achieves all three properties in Zooko's Triangle. Namecoin is the most commonly-used alternative DNS system and is noted for its security and censorship resistance. In 2014,

Namecoin was recognized by ICANN as the most well-known example of a PKI and DNS system with an emphasis of distributed control and privacy, a growing trend in light of the revelations about the US National Security Agency (NSA) by Edward Snowden. [12]

While Namecoin is perhaps the most well-known secure distributed DNS system, it too is not applicable to Tor's environment. One of the main problems is one of practicality: it is unreasonable to require all Tor users to be able to download the entire Namecoin blockchain, which currently stands at 2.05 GB as of January 2015, [13] in order to know DNS records and verify the uniqueness of names. While this burden could be shifted to Tor nodes, Tor clients must then be able to trust the nodes to return an accurate record or even the correct block that contained that record. If the client queried the Namecoin network for DNS information through a Tor circuit, they would have no way of verifying the accuracy of the information without holding a complete blockchain to verify it. Secondly, the hidden service operator would have to prove that he owned both the ECDSA private key attached to the Namecoin record and the private RSA key attached to his hidden service in a manner that is both publicaly confirmable and didn't compromise his identity or location. These problems make Namecoin difficult to integrate securely into Tor. While we recognize Namecoin for its novelty and EsgalDNS shares some similarities with Namecoin, Namecoin itself is not a viable solution here.

Namecoin meets only some of our initial requirements; anonymity during registration can be met when the hidden service operator uses a Tor circuit, anonymity or privacy-enhancement during lookup can be met when the Tor client uses a Tor circuit for the query, public confirmability of the name to the hidden service is not easily met, domain name uniqueness is met by the Namecoin network but not easily proven without access to the entire blockchain, the distributed property is met by the nature of the network, simplicity is not fully met by Namecoin and further software would have be developed to accomplish this objective for Tor integration, backwards compatibility is unknown but could theoretically be met by certain designs, and resistance to malicious modifications or falsifications is met by the network but again not easily proven without access to the entire blockchain.

CHAPTER 6

SOLUTION

6.1 Overview

I propose a new DNS system for Tor hidden services, which I am calling EsgalDNS. *Esgal* is a Sindarin Elvish noun from the works of J.R.R Tolkien, meaning "veil" or "cover that hides". [14] 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 other DNS systems, and its usage is similar to the traditional Clearnet DNS system. 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 Query,s Create, Modify, Move, Renew, and Delete.

6.2 Components

6.2.1 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. [15] 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.

6.2.2 Quorum

As a distributed system, EsgalDNS may have many participants; any machine with sufficient storage and bandwidth capacity — including those outside the Tor network — may perform a full synchronization with the EsgalDNS network and obtain a full mirror of all DNS information. Inside the Tor network, these participants are divided into three main categorizes: *quorum* nodes, quorum node *candidates*, and *mirrors*. *Mirrors* are Tor nodes that have performed a full synchronization and hold a complete copy of all DNS data structures. However, they are either not qualified or have opted out of being a quorum node *candidate*. The requirements to become a candidate are described in the next section. Of those that qualify, the *quorum* is chosen as a random subset from the *candidate* list. The *quorum* perform the main duties of the system, namely broadcasting and recording DNS records from hidden service operators.

Any party at any time may derive the list of *candidate* nodes and the *quorum* from them by the following procedure:

1. Obtain a remote or local archived copy of the consensus document, *cd*, published 00:00 GMT that morning.
2. Extract the digital signature and verify *cd* against $PK_{authorities}$.
3. Construct a numerical list, *ql* of qualified Tor nodes (defined below) 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*.

The consensus document is an authenticated source of information and entropy that can be safely distributed publicly, so all parties — in particular Tor nodes and clients — agree on the members of the *quorum*. As the *quorum* is determined by the consensus document at the beginning of the day, *quorum* nodes have an effective lifetime of 24 hours before they are replaced by a new *quorum*.

Qualifications

It is essential that *quorum* nodes have an up-to-date and complete copy of all DNS data and be ready to accept new information. Of equal importance, all *quorum* members must have sufficient CPU and bandwidth capabilities to handle the influx of new records and the work involved with propagating these records to other Tor nodes. Fortunately, Tor's infrastructure already provides capabilities to easily find the second criteria: Tor nodes receive the *fast*, *stable*, *running*, and *valid* flags if they have demonstrated their 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. These machines meet the second criteria for qualification as a *quorum candidate*.

To meet the first requirement, Tor nodes must demonstrate their readiness to accept new records. The naïve solution to this problem would be have Tor nodes and clients simply ask the node if it was ready, and if so, for the necessary information that demonstrates it. However, this solution quickly runs into the problem of scaling; Tor has 7000 nodes and 2,250,000 daily users [1]: it is infeasible for any single node to handle queries from all of them. Therefore a better solution is to publish information to the authority nodes that all parties can see in the consensus document. To do this, a Tor node who has completed a full synchronization of the system performs the following:

1. Select a *page*-chain according to the rules specified below.
2. Generate a local *NameCache* (described below), *nc* from the *page*-chain.
3. Let s be SHA-384(nc).
4. Encode s to Base64 and truncate to 8 bytes.
5. Wrap the result in parentheses and append the result to the Contact field in the descriptor sent to the authority nodes.

For this last step, 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 portions of any donations to that address, presumably to the Tor relay operator.

The result is appended to the end of the Contact field, and must be updated at least every 24 hours. To become a candidate for membership in the quorum, a node must be a fast and stable node (flags determined and assigned by authority nodes) and must publish this truncated hash that is in the majority of hashes published by other Tor nodes. This can be easily be determined by all parties holding a recent or archived copy of the consensus document.

The generated SHA-384 hash can then be embedded in the consensus document.

To embed the hash, the hash is first converted to base64, truncated to 6 bytes, and surrounded by parenthesis.

6.2.3 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*, *consensusDocHash*, and *nodeKey* of a previous page.

recordList

An array of records, sorted alphabetical or in any other globally deterministic manner.

consensusDocHash

The SHA-384 of the morning's consensus document.

nodeFingerprint

The Tor fingerprint of the quorum 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 of the preceding fields.

Each quorum node holds its page, the pages of the other quorum nodes on that day, and a single archive of the consensus document.

To generate its page, it first picks a page from the past quorums by the following method:

1. Of all chains of pages in the database, ignore any invalid pages and any pages that form chains using those invalid pages. A page may be invalid if it contains records that do not follow the below specifications, its *consensusDocHash* field does not match the hash of the consensus document on that day, or if *pageSig* does not verify. Invalid pages may suggest that the quorum node was not fully synchronized, that it acted maliciously, or that there was data corruption.

2. From these valid chains, find the chain that has been used by the largest number of Tor quorum nodes (not counting today's quorum) and choose the most recent page.

3. If there are multiple chains that satisfy the second condition, choose from among them the page that contains the largest amount of records.

4. If the third condition cannot be resolved, choose from among them the page that is held by the smallest i in the M quorum nodes.

It then takes the SHA-384 of *prevHash*, *recordList*, and *consensusDocHash* of that page, forming *prevHash*. Secondly, it generates *consensusDocHash* by hashing the morning's consensus document. Finally, it hashes the page and generates a digital signature of that hash, storing the result in *pageSig*. The usage of *recordList* is described below.

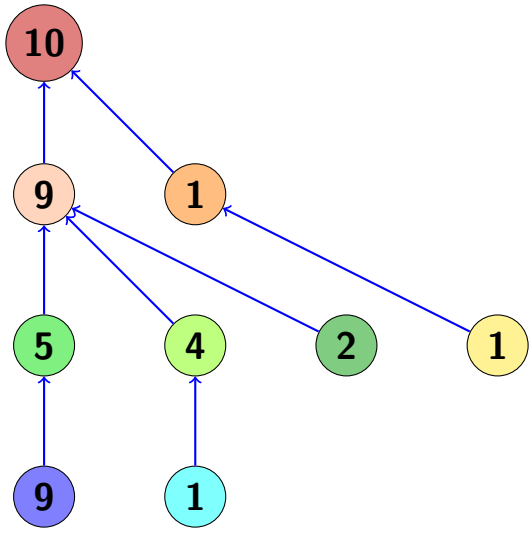

Figure 6.1: Sample page-chain on day 4. Quorum size is $M = 10$, the numbers represent quorum participants with the same page. Despite malicious collusion and misbehaving nodes on day 3, most of day 4's quorum choose the correct chain with 10+9+1+5+4+2 = 31 total participants.

6.2.4 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 quorum nodes, and are only held and processed by quorum nodes in the current day. Snapshots contain three fields: *originTime*, *recentRecords*, and *snapshotSig*, where *originTime* is the timestamp when the snapshot was first created, *recentRecords* is a list of DNS records, and *snapshotSig* is the digital signature of *originTime* and *recentRecords*. When a hidden service operator informs a quorum node about a new record, the quorum node first confirms that the record is valid (described below) and if it is, it adds that record into *recentRecords* and updates *snapshotSig*.

Where $snap_x$ is the current snapshot at propagation iteration x, at each 15 minute mark each active quorum node $node_j$ performs the following:

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. Randomly selects $M/2$ nodes from the M quorum nodes, defining *swapSet*.

3. With each node $node_k$ in *swapSet*, it swaps $snap_x$ and *pageSig* with node $node_k$, receiving a snapshot $s_{x,k}$ and signature $sig_{x,k}$, and then verifies the validity of $s_{x,k}$. For efficiency, if $node_k$ in turn selects $node_j$, no swap will need to take place.

4. After all swapping is complete, $node_j$

   (a) Archives $snap_x$ and each $s_{x,k}$ and $sig_{x,k}$ for all k in *swapSet*.

   (b) Updates its copy of $node_k$'s page by merging in $s_{x-4,k}$, then confirming that $sig_{x,k}$ verifies against the generated page. If not, it asks $node_k$ for its page so that the discrepancy can be resolved.

   (c) Updates its page by merging the contents of $snap_{x-4}$ into *recordList* and regenerates *pageSig*.

   (d) Deletes $snap_{x-4}$.

   (e) Increments x.

TODO: this is incomplete. Quorum nodes send and receive a set of snapshots from other quorum nodes. They pass them along. After 4 rounds of share-with-50-percent the information should have reached all quorum nodes with a high probability, so the information can be safely added into the page and the signature updated. Likewise snapshots from other quorum nodes that are 4 rounds old can be added into its page, hence everyone will be in agreement, theoretically at least. Experiments needed and planned.

6.2.5 Synchronization

Tor nodes must perform a full synchronization with the quorum nodes or node(s) mirroring the database before they are qualified to be picked as part of a quorum. Additionally, they may optionally synchronize in order to become a mirror of the database themselves.

Synchronization occurs in O(n) time, where n is the number of days since since first page was generated. This is possible because each quorum node contains the pages from

all other quorum nodes, so the synchronizing node only has to ask a single node in the quorum for the pages contained by all other sister quorum nodes for each day. Once the node has queried the nodes all the way back to the first quorum it can then follow the selection procedure described above in the Page section.

6.2.6 Registration Creation

Any hidden service operator may claim any domain name that is not currently in use. As domain names cannot be purchased from a central authority, it is necessary to implement a system that introduces a cost of ownership. This performs three main purposes: 1) thwarts potential flooding of system with domain registrations, 2) introduces a cost of investment that improves the availability of hidden services, and 3) makes domain squatting more difficulty, where someone claims one or more 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 introduce a proof-of-work scheme that makes registration computationally intensive but is also easily verified by anyone.

A domain registration consists of nine components: *name*, *subdomains*, *contact*, *timestamp*, *consensusHash*, *nonce*, *pow*, *recordSig*, and *pubHSKey*. The variable *central* consists of all fields except *recordSig* and *pow*.

name

A required field that specifies the domain name that the hidden service operator wishes to obtain, up to 32 characters in length. Domain names uses the .tor TLD.

subdomains

Names of the subdomains used before *name* and their destinations. For example "sample", "a.b.tor" generates sample.example.tor that resolves to a.b.tor. They can also point to .onion addresses or traditional Clearnet domain names. Each subdomain

and their destinations may each be up to 32 characters in length, and up to 16 subdomains may be registered per registration. This is an optional field. *contact*

The fingerprint of the HS operator's PGP key, if he has one. If not or if he chooses to withhold this information, this field is left blank. If the fingerprint is listed, clients may query a public key server for this fingerprint, obtain the operator's PGP public key, and contact him over encrypted email. *timestamp*

The UNIX timestamp of when the operator created the registration and began the proof-of-work to validate it. This is a required field. *consensusHash*

The SHA-384 of the morning's consensus document at the time of registration, encoded in base64. This a provable and irrefutable 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. This is also required. *nonce*

A required field consisting of four bytes that serve as a source of randomness for the proof-of-work, described below. *pow*

16 bytes that demonstrate the result of the proof-of-work. Encoded in base64 and required. *recordSig*

The digital signature of all preceding fields, signed using the hidden service's private key. Encoded in base64 and required. *pubHSKey*

A required field holding the base64-encoding of the hidden service's public key.

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. 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. While this 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 for obvious security

reasons. 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 take 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. The valid and complete record is then represented in JSON format when transmitted and stored by quorum nodes.

TODO: specify how difficulty increases to counteract Moore's Law. Also, is the pow even necessary, or can that be regenerated by anyone?

6.2.7 Record Modification

A hidden service operator may wish to update his registration with more current information. He can generate and broadcast a modification record, which contains updates to the field. The proof-of-work cost is a fourth of registration creation.

6.2.8 Ownership Transfer

Domain names may be transferred to a new owner from one hidden service operator to another. The transfer record contains the public key and signature of the originating owner and the public key of the new owner. The proof-of-work cost is an eighth of generating the domain name in the first place.

6.2.9 Domain Renewal

Domain names expire every 64 days, so they must be renewed periodically. To do this, a hidden service operator generates a renewal record and broadcasts it to the quorum nodes. The proof-of-work cost is a fourth of generating the domain name in the first place.

6.2.10 Registration Deletion

If a hidden service operator wishes to relinquish ownership rights over a domain name,

they can issue a deletion record. This may happen if the hidden service key is compromised, the operator no longer has any use for the domain name, or for other reasons. The deletion record contains the hidden service public key and corresponding digital signature, and once issued to the quorum immediately triggers an expiration of that domain name, making it available to others by a creation record. There is no computational cost associated with a deletion record.

6.2.11 Broadcast

A hidden service operator uses a Tor circuit to contact a quorum node. For security purposes, they must use the same entry node that their hidden service uses. Once the three-hop circuit has been established, they can use their exit node to give a record to the quorum node. That quorum node will add the record to their snapshot, which will be propagated to other quorum nodes and merged into pages for long-term storage.

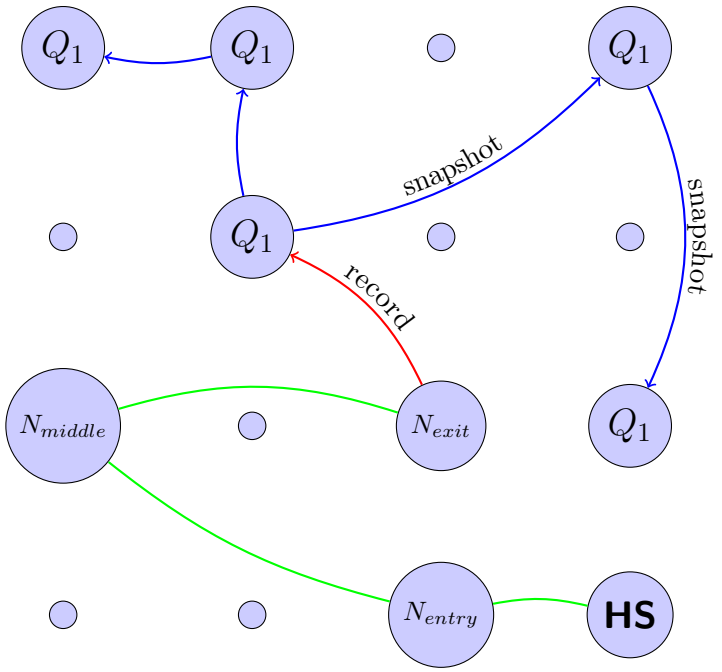

Figure 6.2: HS operator broadcasts a record though a circuit (green) to a quorum node. The record propagates through snapshots.

6.2.12 Registration Query

binary hashtable, fallback alphabetized hashes of names/subdomains

When a Tor user wishes to visit example.tor, his client software must first perform a query to obtain the .onion address that corresponds to that domain name. The client must be able to verify that the received record originated from the desired hidden service, akin to SSL certificates on the Clearnet. Furthermore, the lookup must happen in an anonymous or privacy-enhanced manner and the details of the query must be handled behind-the-scenes, invisibly to the user. These policies satisfy our verifiability, privacy, and simplicity requirements.

At startup, the Tor client software may build a circuit to a quorum node or to any node mirroring the database. He then asks the node server for the domain name. Depending on the verification level specified by the client, this could be limited or large amounts of information. As we mentioned in the requirements section, it is impractical to require the client to download the entire DNS database to perform verification, so instead the client can perform various degrees of verification with minimal information.

At level 1, the server returns just returns the creation record. The client software then extracts the fields, uses *pubHSKey* to confirm that *recordSig* verifies the record, then confirms the proof-of-work. The client then uses *pubHSKey* to generate the hidden service .onion address (eight bytes of the base32-encoded SHA-1 hash of the PKCS.1 DER encoding of *pubHSKey*) and looks up the hidden service in the traditional manner. The lookup fails if the service has not published a recent hidden service desciptor to the distributed hash table, otherwise the lookup goes through. 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*.

At level 2, the server return the record, the page that it was included in, the *pageSig* from all quorum nodes on that day, and an archive of the consensus document. This is *width verification*; the client can verify the authenticity of the consensus document, determine the old quorum members and their keys, hash the page, and then verify that all the *pageSig*s

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

Figure 6.3: Sample empty initial page, $p_{1,1}$

verify against that hash. The client then proceeds with record verification and hidden service lookup, as specified in level 1.

At level 3, the server return the record, the page that contained that record, and the pages that form the chain below it. This is *depth verification*; the client can confirm the length of the page-chain and records contained within. This is demanding both in computation and in bandwidth usage, but is an extremely thorough level of verification.

TODO: at level 4, include consensus documents so that the old quorums can be found?

At all levels, the server may return the introduction point for that hidden service, which the node can determine in advance by querying the distributed hash table and then cashing the result. This would significantly improve performance because the client can skip that step.

6.3 Examples and Structural Induction

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 simple. On day_0, c_1 generates an initial page $p_{1,1}$ containing no records and signs $p_{1,1}$, but does not accept records for this initial page. For reference, $p_{1,1}$ appears as

On day_1, c_1 examines its database of page-chains and generates a new page, $p_{2,1}$, that

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

Figure 6.4: Sample registration record from a hidden service. Note that the "sub.example.tor" → "example.tor" → "exampleruyw6wgve.onion" references can be resolved recursively.

references $p_{1,1}$, a chain with 0 references, the most in the database. The hidden service *Bob* hashes the consensus document, generating *T/q7q052MgJGLfH1mBGUQSFYjwVn9VvOWBoOmevPZgY=* 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 as:

c_1 adds this record to its snapshot, creating

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, nothing need happen here. c_1 then adds r_{reg} into its page, creating $p_{1,1}$:

```
{
    "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/9
        FaN6ZjyHAnvjmSpRRSngR0UD20FwpAZ1vCVA0qO2yDZeuBd6DiNSkkdSueRHOF7OD95Rb04JmAk1jXjF
        +BH3hUH54ZEaqlJvQ8tBQJ7YtAc="
}
```

Figure 6.5: Sample snapshot from c_1, containing one registration record r_{reg} from a hidden service.

This record → snapshot → 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 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 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.

```
0  {
1      "prevHash": 0,
2      "recordList": [
3          {
4              "names": {
5              "example.tor": "exampleruyw6wgve.onion",
6              "sub.example.tor": "example.tor"
7          }
8          "contact": "AD97364FC20BEC80",
9          "timestamp": 1424045024,
10         "consensusHash": "uU0nuZNNPgilLlLX2n2r+sSE7+N6U4DukIj3rOLvzek=",
11         "nonce": "AAAABw==",
12         "pow": "4I4dzaBwi4AIZW8s2m0hQQ==",
13         "recordSig":      "KSaOfzrXIZclHFcYxI+3jBwLs943wxVv3npI5ccY/
               kBEpyXRSopzjoFs746n0tJqUpdY4Kbe6DBwERaN7ELmSSK9Pu6q8QeKzNAh+
               QOnKl0fKBN7fqowjkQ3ktFkR0Vuox9WrrbNTMa4+
               up0Np52hlbKA3zSRz4fbR9NVlh6uuQ=",
14         "pubHSKey": "MIGhMA0GCSqGSIb3DQEBAQUAA4GPADCBiwKBgQDE7CP/
               kgwtJhTTc4JpuPkvA7Ln9wgc+
               fgTKgkyUp1zusxgUAn1c1MGx4YhO42KPB7dyZOf3pcRk94XsYFY1ULkF2+
               tf9KdNe7GFzJyMFCQENnUcVXbcwLH4vAeiGK7R/nScbCbyc9LT+
               VE1fbKchTL1QzLVBLqJTxhR+9YPi8x+QIFAdZ8BJs="
15         }
16     ],
17     "consensusDocHash": "T/q7q052MgJGLfH1mBGUQSFYjwVn9VvOWBoOmevPZgY=",
18     "nodeFingerprint": "2FC06226AE152FBAB7620BB107CDEF0E70876A7B",
19     "pageSig": "KO7FXtoTJmxceJY1W202c0WwRGRyU9m99IskcL9yv/
           wFQ4ubzbjVs8LQzwQub9kDJ8Htpc9rRZvneRRbusFv1nvaeJw+WgRt+
           Tck0uapndHKYaQcK3XTIFYdmT1lLm7QxSKjnIxgBkwKT0QWdGLUhuRgGe5CXmqrPeDfU
           /gsgLs="
20 }
```

Figure 6.6: c_1's page, containing a single registration record.

6.4 Fault Tolerance

Tor nodes have no reliability guarantee and may disappear from the network momentarily or permanently at any time.

CHAPTER 7

ANALYSIS

general analysis...

7.1 Security

7.1.1 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 [16]) 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.

Requirements for stable and fast nodes makes spinning up lots of nodes (LizardSquad) more challenging.

7.2 Performance

bandwidth, CPU, RAM, latency for clients..

7.2.1 Load

demand on participating nodes...

CHAPTER 8

RESULTS

8.1 Implementation

implementation...

We use the BSD-licensed Botan library and C++11 for the hash and digest algorithms in our reference implementation, while the Mersenne Twister is implemented in C++'s Standard Template Library.

Our reference implementation uses the libjsoncpp header-only library for encoding and decoding purposes. The library is also available as the libjsoncpp-dev package in the Debian, Ubuntu, and Linux Mint repositories.

8.2 Discussion

discussions...

8.2.1 Guarantees

Guarantees...

CHAPTER 9

FUTURE WORK

Some open problems that I need to address include:

1. How frequently should domains expire? Are there any security risk in sending a Renew request?

2. How should unreachable or temporarily down nodes be handled? I'd like to know the percentage of the Tor network that is reachable at any given time.

3. How many nodes in the Tor network should be assumed to be actively malicious? What are the implications of increasing this percentage?

4. What attack vectors are there from the committee nodes, and how can I thwart the attacks?

5. What are the implications of a node intentionally voting the opposite way, or ignoring the request altogether?

6. What would happen if a node was too slow or did not have enough storage space to do its job properly?

7. What other open problems are there?

8. What related works are there in the literature that relate to the concepts I have created here?

CHAPTER 10

CONCLUSION

I have presented EsgalDNS, a distributed DNS system inside the Tor network. The system correlates one-to-one human-meaningful domain names and traditional Tor .onion addresses. Tor nodes and clients can both verify the authenticity, integrity, and uniqueness of registrations. Although this design is nowhere near finalized, so far this system seems both promising and novel. I have more work to do, and I am planning to implement and test this system on a simulated Tor network. If accepted by the Tor community, I believe that EsgalDNS will be a valuable infrastructure that will significantly improve the usability and popularity of Tor hidden services.

REFERENCES

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

[2] T. Guardian, "Tor: 'the king of high-secure, low-latency anonymity'," http://www.theguardian.com/world/interactive/2013/oct/04/tor-high-secure-internet-anonymity, 2013, [Online; accessed 4-Feb-2015].

[3] L. Xin and W. Neng, "Design improvement for tor against low-cost traffic attack and low-resource routing attack," *2009 International Conference on Communications and Mobile Computing*, 2009.

[4] Z. Ling, J. Luo, W. Yu, X. Fuc, W. Jia, and W. Zhao, "Protocol-level attacks against tor," *Computer Networks*, 2012.

[5] D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker, "Shining light in dark places: Understanding the tor network," *Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2969*, 2008.

[6] S. Nicolussi, "Human-readable names for tor hidden services," 2011.

[7] M. S. Ferdous, A. Jsang, K. Singh, and R. Borgaonkar, "Security usability of petname systems," October 2009.

[8] M. Stiegler, "Petname systems," 2005.

[9] katmagic, "Shallot," https://github.com/katmagic/Shallot, 2012, [Online; accessed 4-Feb-2015].

[10] N. Mathewson, "Next-generation hidden services in tor," https://gitweb.torproject.org/torspec.git/tree/proposals/224-rend-spec-ng.txt, 2013, [Online; accessed 4-Feb-2015].

[11] K. Okupski, "Bitcoin developer reference," 2014.

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

[13] bitinfocharts.com, "Crypto-currencies statistics," https://bitinfocharts.com/, 2015, [Online; accessed 4-Feb-2015].

[14] D. Willis, "Hiswelk's sindarin dictionary, edition 1.9.1, lexicon 0.9952."

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

[16] 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.