TrickleDNS: A Safety Net for the Domain Name System

Sriram Sankararaman* Venugopalan Ramasubramanian† Lakshminarayanan Subramanian‡ and Ion Stoica*

*University of California Berkeley, Berkeley, CA 53706
†Microsoft Research Silicon Valley, Mountain View, CA 94043
†New York University, New York, NY 10003

October 9, 2007

Abstract

This paper presents TrickleDNS, a practical and decentralized system for disseminating DNS data securely. Unlike prior solutions, which depend on the as-yet-undeployed DNSSEC standard to preserve data integrity, TrickleDNS uses a novel security framework that provides resilience from data corruption by compromised servers and denial of service attacks. It is based on the key design principle of randomization: First, TrickleDNS organizes participating nameservers into a *well-connected* peer-to-peer network with random yet constrained links to form a *Secure Network of Nameservers (SNN)*. Nameservers in the SNN reliably broadcast their public-keys to other nameservers without relying a centralized PKI. Second, TrickleDNS *reliably binds domains* to their authoritative name servers through independent verification by multiple, randomly chosen peers within the SNN. Finally, TrickleDNS servers *proactively disseminate* self-certified versions of DNS records to provide faster performance, better availability, and improved security. This paper validates TrickleDNS through simulations and experiments on a prototype implementation.

1 Introduction

The Domain Name System (DNS) forms a critical component of the Internet infrastructure by providing the essential service of host name to IP address resolution. Internet users and providers of web-based services implicitly assume and rely on the correct operation of DNS. However, DNS as operated today is susceptible to a wide range of attacks that enable malicious elements to hijack domain traffic by propagating bogus address mappings and to make the domain unavailable by disrupting name resolution. These vulnerabilities primarily stem from limited redundancy in DNS; recent studies show that 80% of domain names are served by just two name servers, while 32% of domain names have all name servers behind the same network gateway [24, 26]. And the risks are further exacerbated because of the trust placed unwittingly by many domain operators on name servers outside the domain's control [27].

The prevalent solution to tolerate malicious attacks and achieve strong data integrity in DNS is DNSSEC [2]. It enables clients to verify the authenticity of a domain's data with the help of a chain of certificates signed by the domain and each of its parent domain, eventually attested by the public key of a single, globally-known root domain. However, this dependency of DNSSEC on a centralized Public Key Infrastructure (PKI) has significantly limited the acceptance of DNSSEC despite many years of efforts. Meanwhile, individual domains are unable to secure themselves until all their parent domains adopt DNSSEC. Furthermore, DNSSEC does not improve the resilience of DNS against Denial of Service (DoS) attacks.

This paper presents TrickleDNS, a completely decentralized approach for securing the DNS against threats to data integrity and availability. Unlike DNSSEC, TrickleDNS does not rely on a PKI or a trusted central authority. Instead, it acquires its security through the following key concepts:

Secure Network of Nameservers: TrickleDNS establishes secure distribution of domain public keys through a random, well-connected network of name servers. This network provides a sufficient number of redundant paths between any two name servers so that they can exchange public keys reliably even in the presence of a sizeable number of compromised servers. Two techniques enable TrickleDNS to achieve high resilience to attacks: First, servers are forced to choose their neighbors in an "explicitly constrained" manner in order to restrict the freedom of malicious agents. These constraints allow the TrickleDNS overlay with $O(\log n)$ neighbors per server to tolerate $O(\frac{n}{\log n})$ number of compromised servers. Second, adversaries with large number of identities are restricted from launching Sybil attacks [14] by forcing them to pick

identities from different IP prefixes (say /24). Thus, Sybil attackers need to own several blocks of IP addresses in order to be effective.

Reliable Name Binding: TrickleDNS binds domains to their authoritative name servers through independent verification by multiple, randomly-chosen servers in the network. The verification relies on the delegation (NS) records at the parent domain. While this process is trivial if the parent domain already participates in TrickleDNS, for other domains, servers verify the existing parental chain of delegation records in DNS. Even though this process places trust back on the legacy DNS to bootstrap itself, we believe that it is a small price to pay in order to provide secure data dissemination for domains whose parents are not covered by DNSSEC. Moreover, higher-level domains tend to be more secure than lower-level domains as shown in Section 5.1.3.

Proactive Dissemination of DNS Data: Finally, TrickleDNS servers push-out self-certified versions of DNS data on the secure network in order to improve lookup performance and availability. The dissemination mechanism keeps the overhead small by broadcasting only a critical set of records (NS records, the corresponding glue A-records, and SOA records) as proposed by Handley et al. [18]. It reduces lookup latency substantially (a typical name resolution in TrickleDNS just involves one query to the authoritative server) and mitigates denial-of-service attacks.

TrickleDNS is by no means a complete replacement for the current DNS. Instead, it is meant to serve as a safety-net, improving the resilience of current DNS against domain hijacks and DoS attacks. Certainly, a centralized security solution such as DNSSEC if and when deployed widely would obviate the need for some of the mechanisms presented here. However, that date for a complete adoption of DNSSEC seems to be nowhere in sight. Meanwhile, we expect that a decentralized security model such as TrickleDNS will meet the need for thousands of vulnerable domains.

TrickleDNS is easy to adopt. Despite relying on a cooperative network of servers, trust is not placed on any server outside one's own domain. It tolerates a large number of colluding malicious servers as well as attackers with multiple identities. It bootstraps from existing DNS records (through a careful verification based on majority consensus) while taking advantage of centrally-rooted certificate chains when available; it thus facilitates a scenario where TrickleDNS coexists with a partially-deployed centralized security scheme. Overall, this impels a quick and incremental deployment of TrickleDNS driven by personal incentives of each domain.

This paper makes the following key contributions. First, it presents the design and implementation of TrickleDNS as outlined above. Second, it provides a detailed simulation-based analysis of the security properties of TrickleDNS. And finally, it reports on performance measurements of our prototype implementation showing a factor of 10 improvement in the median query latency for memory and bandwidth overheads that are well within the capacities of modern computers and networks.

2 DNS: Background and Related Work

The Domain Name System (DNS) is a general-purpose database for mapping names from a globally unique name space to data resources associated with a name. It uses a hierarchical name space partitioned into non-overlapping regions called *domains*. For example, *foo.bar.com* is a sub-domain of *bar.com*, which in turn is a sub-domain of the top-level domain *com*, which is under the global root domain. Resources for names within a domain are served by a set of nodes called the *authoritative name servers*. In addition to network addresses for host names, DNS resources, called *records*, could also include names of authoritative DNS servers, names of mail servers, or any small-sized data associated with the domain.

DNS uses a delegation based architecture for name resolution [22, 23]. Clients resolve names by following a chain of authoritative servers, starting from the root, followed by the top-level domain name servers, down to the servers of the queried domain. For example, the name *www.foo.bar.com* is resolved by following the authoritative severs of the parent domains *com*, *bar.com*, and *foo.bar.com*. DNS lookups could take a long time to follow the chain of servers in the hierarchy. To improve the lookup latency, DNS servers aggressively cache responses.

2.1 Threats to DNS Security

DNS clients implicitly trust all the servers involved in the resolution of a name and cannot verify the authenticity of records. A malicious agent successful in compromising one or more DNS servers can easily introduce bogus records with IP addresses pointing to destinations operated by the malicious agent. For instance, the web site of RSA, an Internet security firm, was recently hijacked by compromising its DNS servers [1]. We use the term *data integrity violation* to describe the attacks that result in domain hijacks. Alternatively, they could make all services hosted by the domain unavailable by disrupting name resolution. In this section, we describe the key vulnerabilities in DNS that facilitate data integrity violations and DoS attacks.

Data Integrity Violations

An attacker can compromise data integrity in today's DNS in several different ways. First, it can directly compromise an authoritative name server of a domain and

use it to disseminate malicious records. Second, it can compromise the DNS servers used to locate the authoritative name servers thereby redirecting queries to malicious servers. Third, it can exploit loopholes in the DNS caching algorithms to introduce malicious records. Vulnerabilities in DNS facilitate each of the above three attacks.

Compromising name servers: Implementations of DNS servers have serious software bugs such as buffer overflows that can be exploited to take over the server. For instance, BIND, the widely-used implementation of DNS, has at least twenty documented security vulnerabilities [7]. Of which, at least one affects about 17% of DNS servers [26]. The high prevalence of software vulnerabilities further exacerbates the already limited redundancy in authoritative name servers. The study by Ramasubramanian and Sirer [26] showed that about 80% of the surveyed domains had only two authoritative servers and a small 1% only one. A similar DNS survey by Pappas et al. found that 65% of the surveyed domains had only two authoritative servers.

Perils of transitive trust: DNS uses name-based delegations, where authoritative name servers are described by their host names instead of IP addresses. The delegation based architecture implies that resolving a domain name triggers additional name resolutions to get the address of the authoritative servers. Such delegations force each domain to depend on a large number of servers, which can be distributed among different domains. For instance, *cornell.edu* depends on a server in *rochester.edu*, which depends on *wisc.edu* and in turn on *umich.edu*, with a total dependence on 48 servers in 20 different domains. An average domain name depends on 46 servers while names in several country-code domains depend on more than 100 servers [27]. Such subtle dependencies between name servers administered by different domains make DNS extremely difficult to secure. An attacker can compromise any server that a domain depends on and misdirect the DNS requests passing through that server.

Cache poisoning: Cache poisoning is a way to introduce fake records by exploiting DNS servers' propensity to cache records without verifying the authority of the host providing the record [15, 4, 29]. It typically occurs when DNS servers cache 'glue' records, which are the address records associated with name server (NS) records, added to DNS responses to enable name resolution. Nearly $25 - 30\%$ of DNS servers are estimated to be vulnerable to cache poisoning attacks [13]. Fortunately, cache poisoning can be easily fixed by not caching records obtained from non-authoritative servers. Several implementations of DNS servers, including *djbdns* [6], are robust against cache poisoning attacks.

Denial of Service Attacks

The hierarchical structure of DNS makes it highly susceptible to denial of service (DoS) attacks. In particular, servers at the upper levels of the hierarchy receive a greater portion of query load and constantly face denial of service attacks [8, 9]. The most widely-known DoS attack on DNS root servers occurred in October 2002 affecting 9 of the 13 root servers and interrupting DNS services briefly. As a response to this attack, the DNS root is currently served by more than sixty servers using BGP-level anycast to dynamically balance load between them.

However, the lower levels of DNS hierarchy continue to be highly vulnerable to targeted DoS attacks due to limited redundancy. The risk of DoS attacks is further exacerbated by the fact that a large number of domains, about $30 - 70\%$, have all their name servers behind a single shared gateway [24, 26]. A link saturation attack on the gateway or a benign network failure could easily make the domain unavailable.

2.2 DNSSEC

DNSSEC [13, 2, 15, 29] is the foremost solution that addresses the data integrity issues in DNS. DNSSEC relies on public-key cryptography to generate certificates and verify authenticity. Each record belonging to a domain has a certificate signed by the domain's private key, while its public key is disseminated through DNS as a key record. In order to prove ownership of its name space, the domain obtains a certificate from its parent domain, which consists of its key record signed by the parent's private key. Essentially, DNSSEC associates each domain with a chain of certificates signed by the centralized root all its parent domains. DNS clients are seeded with the public key of the root domain and can verify the integrity of records by following the chain of certificates.

Although DNSSEC appears to be the elixir for preserving data integrity in DNS, its acceptance has been remarkably poor. Despite its presence for more than ten years, its specifications are continuously modified as new vulnerabilities in the spec are discovered. Currently, very few top level domains (notably *.se*) support DNSSEC. The inability to support an incremental, bottom-up deployment model means that a domain that wishes to secure itself is unable to use DNSSEC unless all its parent domains adopt DNSSEC. Moreover, DNSSEC relies on the same hierarchy of servers and is susceptible to DoS attacks.

2.3 Related Work

Improving Data Integrity: A few researchers have proposed techniques to improve DNSSEC. Ateniese and Mangard [3] propose to use symmetric key cryptography to reduce the computational overhead of DNSSEC.

Massey et al. [21] propose to use decentralized certificate chains, which consist of certificates signed by peer domains. An example certificate chain would consist of *yahoo.com*, endorsing *microsoft.com*, which endorses *google.com*, etc. There are two problems with this approach: first, a domain has no control over the head of its certificate chain, that is, even a malicious domain can create legitimate certificate chains, and second, it is difficult to discover certificate chains without performing a flooding based search.

There are also several proposals to improve DNS data integrity without relying on DNSSEC. Fetzer et al. [16] propose use OpenSSL as a trusted third party to generate certificates for DNS records. Their proposal also uses certified PKI and has the same limitation as DNSSEC. Yang [33] proposes to replace name servers with practical byzantine fault tolerant systems. Wang et al. [32] as well as Cachin and Samar [10] propose to apply threshold cryptography for securing the signing process.

ConfiDNS [25] alleviates security problems stemming from the trust placed on arbitrary servers by looking for agreement among data fetched from different servers. It is a client-driven solution that is intended to safeguards users from failures of client-side DNS infrastructure.

Alleviating DoS Attacks: Several researchers have proposed the use of self-organizing peer-to-peer overlays to achieve high resilience against failures and DoS attacks. DDNS [11] implements legacy DNS functionalities on top of the Chord structured overlay [30]. Overlook [31] is a new name service layered on top of the Pastry structured overlay [28]. CoDoNS [26] employs proactive, analysis-driven caching on structured overlays to provide a name service with low lookup latencies and high failure resilience. Recently, Handley et al. [18] proposed an architecture to push DNS records using a peer-to-peer overlay for high performance and failure resilience. While the above proposals provide high resilience against DoS attacks, they expand the trusted computing base to include all nodes in the system. Consequently, these proposals rely on the deployment of DNSSEC or an alternative PKI to preserve data integrity.

In addition, there have been proposals to serve the entire DNS data from a single, centrally-managed repository [20, 12]. Deegan et al. [12] argue that the entire DNS query load can be handled by a just few hundred servers. While centrally managed systems can provide good performance and availability just like the centrally managed root servers today, they do not obviate the need for a security solution to verify data integrity.

3 TrickleDNS

Figure **??** illustrates the architecture of TrickleDNS. TrickleDNS is a push-based DNS, which proactively propagates critical DNS records of participating domains to all servers so as to provide better availability, lower query resolution times, and faster update propagation. It pushes DNS records in a cooperative, peer-to-peer network formed by the authoritative nameservers of participating domains. A domain can join TrickleDNS even if its parent is not a participant. Each participating nameserver, called a *trusted nameserver* or TN, has a certain relatively small number of other TNs as neighbors. Together, these TNs along with their direct neighbors form a distribution network called the *Secure Nameserver Network (SNN)*. The TNs use the SNN to ditribute DNS records among themselves. Each TN disseminates the DNS records of the domain it represents to its neighbors and additionally, also forwards records it receives from its neighbors. Later in this Section, we will elaborate as to how the SNN is created and how each TN can distribute DNS records in a secure and verifiable manner within the SNN.

The trusted nameservers also interact with three other external entities that are part of the current DNS: authoritative nameservers of non-participating domains, local resolvers that resolve names on behalf of clients, and the clients themselves. Since we strive for incremental deployment, our design does not impose any changes to the operation of these external DNS entities. TrickleDNS supports the same query interface as existing DNS and interacts with non-participating nameservers through standard DNS protocols. Additionally, local name resolvers can be enabled to receive broadcast data from the TrickleDNS network without participating in data dissemination.

3.1 Security Threat Model

The key challenge in TrickleDNS is to ensure that the propagation of DNS records is secure from attacks by adversaries. Adversaries might be interested in targeting one or more specific domains or attacking the entire system. We do not focus on attacks targeted at specific client sites; other researchers have proposed solutions to address client side security [25]. TrickleDNS tolerates the following types of attacks:

1. **Server Compromises**: An adversary might compromise one or more TNs and make it behave in a Byzantine manner, that is, generate bogus records, suppress genuine records, as well as, collude with other compromised TNs to launch strategic attacks. TrickleDNS targets to tolerate a large number of compromised servers, at least a sizeable fraction of the total number of participating servers.

2. **Identity Attacks**: An adversary might directly introduce malicious nameservers into the system compromising proactive dissemination. A serious concern in using a cooperative, peer-to-peer network is the Sybil attack [14],

where an adversary introduces a substantially large number of nameservers and easily takes control of the entire system. We currently tolerate a subset of Sybil attacks where the adversaries have a limited set of IP address blocks from which they draw their identities. More powerful adversaries that have access to a Botnet with widely-distributed IP addresses are not addressed by our current design. Section **??** briefly discusses a few precautionary measures that reduce the risk of Botnet attacks.

3. **DoS Attacks**: Finally, adversaries might launch denial of service attacks on the system through link saturation attacks, excessive querying of targeted TNs, as well as trying to take advantage of TrickleDNS protocols and mechanisms.

3.2 TrickleDNS basic approach

The basic design of TrickleDNS to deal with the above mentioned security threats can be categorized intro three parts:

Decentralized Key Distribution: Every participating TN within TrickleDNS generates its own pair public-private key pair. To provide data integrity of DNS records in the face of adversarial TNs, the first step involves each TN distributing its public key in a decentralized manner to all the participating TNs. TrickleDNS uses two basic steps to achieve this property. First, a carefully constructed TrickleDNS network called *Secure Nameservers Network* (SNN) ensures that nodes are sufficiently randomly distributed and explicitly curtails the power of colluding.adversarial nodes from introducing fake edges in the SNN. The SNN construction also has inbuilt mechanisms to curtail identity based attacks. Second, TrickleDNS leverages ideas from our prior work on *reliable communication toolkit* [**?**] which provides a mechanism that relies on independent vertex-disjoint patha to achieve decentralized key distribution in static networks where nodes have fixed identities. TrickleDNS extends these ideas to P2P environments and can also deal with a much larger number of adversaries as elaborated later in this section.

Reliable Name Binding: Every domain in TrickleDNS, say $foo.bar$, picks a public-private key pair and uses it to create signed certificates for its records. Once decentralized key distribution within the SNN is established, every TN can reliably broadcast the public-key corresponding to the domain it serves to all participating TNs within the SNN. A separate process called *reliable name binding* verifies that a TN signing a domain's public key is indeed its authoritative nameserver.

Push-based Dissemination of DNS records: Once the keys corresponding to the TNs and the domains have been distributed, TrickleDNS pushes DNS records within the SNN. For domains within the SNN, proactive dissemination eliminates the need for the hierarchical lookup as in the current DNS. This is critical to improving DoS resilience of the system.

The rest of this section describes these three aspects in greater detail.

3.3 Decentralized Key Distribution

Each TN s generates a private-public key pair $(s.k, s.K)$ independently. We call the tuple $s.kid = (s.id, s.K, s.seq)$ the *keyed identity* of s, where $s.id$ is the id for s and $s.seq$ is a sequence number used to mark the latest public key. We chose the identifier of a TN to be a collision-resistent function of its IP address since IP address is the unit of identification of a nameserver in DNS. We discuss the implications of this choice for the identifier in Section **??**.

The immediate goal of the key distribution process is to ensure that each TN correctly learns the keyed identity of all other TNs.

3.3.1 Secure Nameserver Network (SNN)

A reliable way to distribute keyed identities to all TNs in the absence of a centralized certification authority is by forming a well-connected distribution network—a network with sufficient independent paths to send information between TNs in order to overcome the influence of potentially malicious or compromised servers. It is well known that a network with at least $2k + 1$ independent, vertex-disjoint paths between each pair of TNs can tolerate up to k malicious servers [**?**]. Two paths are vertex-disjoint if no intermediate server appears on both the paths.

Prior work [**?**] shows that random, peer-to-peer networks, where each server is connected to a fixed number D of other randomly chosen servers, provides an efficient way of building well-connected networks. More precisely, such a random network of neighbor degree D is guaranteed to have at least D vertex-disjoint paths between any two servers with high probability.

However, a totally random network, where participating TNs have the freedom to chose any other TN as their neighbor, has limited attack resilience. This freedom means that an attacker can connect a set of colluding servers as neighbors of a targeted nameserver T and then corrupt any data, including the public key, published by T. Therefore, the network needs to have a minimum neighbor degree $D = 2k + 1$ neighbors per server in order to be resilient to k compromised servers. If this approach were used, the neighbor degree D would have to be an unpractically large number for TrickleDNS since it aims to tolerate compromises to a sizeable fraction of the network.

TrickleDNS overcomes this by imposing constraints on the choice of neighbors of each TN. That is, the TN is

not allowed to chose any other TN as neighbor, but instead its neighbors are chosen as a random, deterministic function of the TN's identity. Thus, a compromised server cannot connect to targeted TNs in the SNN topology; similarly, even though two colluding compromised servers can communicate out-of-band, they cannot pretend to be direct neighbors in this topology.

TrickleDNS achieves a random yet deterministic neighbor selection through the use of *consistent hashing* [**?**]. One way to use consistent hashing is to have an identifier $s.id$ for each TN s drawn from a circular key space. Then, the TN s can pick its i^{th} neighbor as that TN whose identifier is closest to $h(s.id|i)$ clockwise on the key space, where the operation $|$ refers to concatenation and h is a collision resistent hash function whose range is the same circular key space.

Choosing neighbors through consistent hashing enhances the attack resilience in two ways: First, an attacker needs to compromise or introduce a large number of servers in the system before it can control a sufficient number of neighbors of a targeted nameserver. Second, any attempt by the attacker to fake a neighbor relationship with a targeted nameserver will be discovered by other TNs since consistent hashing provides easily-verifiable, deterministic neighbor relationships.

Indeed, the above constrained random network increases the attack resilience of reliable communication from just a small number k of malicious servers, shown in [**?**], to a sizeable fraction of the total network. More precisely, we prove the following theorem in **??**.

Theorem 1. *Reliable communication can be achieved with high probability between any pair of non-malicious nameservers in the presence of* $O(\frac{n}{\log n})$ *malicious nodes provided the node degree* $D \geq \alpha \log n$ *for some* $alpha > 6 \ln 2 \sim 4.15$ *and the paths used for reliable communication are of length at most* $\log n$.

The main crux of the arugment in the result stated above is that: beyond a certain minimum degree for every node in the SNN, there are sufficient number of "short independent" vertex disjoint paths between every pair of nodes such that a majority of these paths do not have any adversarial nodes with high probability. The above result holds under the assumption that adversarial nodes are randomly distributed and the notion of constrained links in this topology tries to enforce randomization in the link selection process.

3.3.2 Handling Identity Attacks

The above manner of using consistent hashing, however, still provides attackers one avenue to choose which part of the network its servers belong. An attacker might choose the IP address of one of its server s in such a way that its hash, that is $s.id$, makes it a broadcast neighbor of the targeted nameserver T.

TrickleDNS limits this attack by using a shorter prefix of the IP address, such as a /24, while assigning broadcast neighbors[1]. This technique ensures that an adversary, which might own a large block of IP addresses as most domains typically do, does not cycle through its available set of IP addresses to connect to a targeted TN. Moreover, in the event that the adversary introduces a large number of nameservers from the same block, they all connect to the same neighbors limiting their impact on proactive dissemination process. This technique, however, cannot tolerate more powerful adversaries that have access to a network of bots with widely-distributed IP addresses. We discuss the implications of Botnet-based attacks in Section **??**.

More precisely, neighbor selection in TrickleDNS works as follows: Each TN s has an identifier $s.id$ in a circular key space obtained by hashing its IP Address; $s.id = \text{SHA-1}(s.ip)$. The TN also has a secondary identifier called *prefix identifier*, $s.id' = \text{SHA-1}(ip_{0\ldots b-1})$, that is, the hash of the first b bits of the IP address. We currently pick a single reasonable value for $b = 24$ called the *maximum spoofable unit* (MSU). Note that many TNs might share the same prefix identifier.

Then, each TN connects to $\mathcal{B}$ other TNs called *broadcast neighbors*, where the i^{th} broadcast neighbor of s is determined by the following two-step process: 1) It finds the set of TNs whose prefix identifier is closest to SHA-1$(s.ip|i)$ clockwise on the key space. Here, the operation $|$ refers to concatenation. 2) From this set of TNs it picks the broadcast neighbor to be that TN t whose primary identifier $t.id$ is clockwise closest to SHA-1$(s.ip|i)$. Note that the second step randomly picks a suitable server from the set of TNs ensuring a uniform, random neighbor relationships even if a disproportionately large number of servers happen to share the same prefix.

3.3.3 Reliable Key Broadcast

Once the SNN is setup, key distribution follows the same protocol described in [**?**]. We give a brief overview of the this protocol for completion.

State: Each TN s stores an *identity graph* G_s with distinct keyed identities it learns and the neighbor relationships between them. Note that there might be more than one keyed identities for the same server either because the server generated a new public key with a higher sequence number or because a malicious server created a fake keyed identity for it. The identity graph enables the TN to verify the authenticity of a keyed identity $t.kid$ by

[1] similar to the approach followed in [17] for peer selection.

checking whether there are at least $\lceil \frac{\mathcal{B}}{2} \rceil$ vertex-disjoint paths between r and t in G_s. If two keyed identities for the same server t passes this check then it accepts the keyed identity with the greater sequence number. One way to perform the disjoint-path check is by running a standard Max-Flow algorithm.

Protocol: TNs exchange their keyed identity through the broadcast of *signed path vector* messages to their neighbors. A TN s sends a signed path vector (SPV) $spv[(s.id, s.K, s.seq), s.b_i.id]_{s.k}$ to its broadcast neighbor $s.b_i$. The SPV contains the keyed identity of the sending TN and the identity of the receiving broadcast neighbor; the whole path is signed using the TN s's private key. Any other TN can verify that the SPV has not been corrupted using the embedded public key.

The receiving server $r = s.b_i$ then extends the SPV by adding its own keyed identity and the identity of the broadcast neighbor to which it will further propagate, signs the extended SPV $spv[spv[(s.id, s.K, s.seq), s.b_i.id]_{s.k}, (r.id, r.K, r.seq), r.b_j.id]_{r.k}$ with its private key, and propagates it. As a shorter form, we denote an SPV that traverses through servers $s_1, \ldots, s_n$ as $SPV = spv[(s_1, \ldots, s_n)]$.

A receiving TN r rejects an SPV under three conditions: First, the SPV has bogus link relationships; that is, the link relationships do not obey the consistent-hashing-based neighbor selection rules. Second, the SPV contains no new link information about keyed identities. And, finally, the SPV does not verify itself; that is, some signature in the SPV does not match the corresponding public key. If the SPV passes these checks, it is scheduled to be further propagated to the broadcast neighbors.

The first condition permits a node to reject a bogus message as soon as it knows that it is bogus. The second condition is a stoping criterion to key the broadcast overhead bounded. The third condition performs a cryptographic verification of the message. Note that signed messages form a critical role in preventing a malicious server from faking keyed identities. That is, in order to falsely speak for a targeted server t, a malicious entity also needs to generate a separate public key for server t creating a new keyed identity for t as well. The false keyed identity should fail the disjoint path check and not be accepted as reliable.

Overhead Analysis: The above broadcast protocol is quite light-weight. We can analyze its message overhead in the steady state where all TNs have learnt a stable topology and a new TN joins the system. The new TN s creates $\mathcal{B}$ new neighbor relationships. Since a TN does not forward an SPV if does not contain any new neighbor relationships, each TN forwards an SPV at the most once for each neighbor relationship to each of its neighbors. As a result, a TN must perform $O(\mathcal{B}^2)$ verifications and transmit $O(\mathcal{B})$ SPVs for each new TN joining the system. The overhead might be higher if several TNs join the system simultaneously. However, we expect large simultaneous joins to be rare in practice. More seriously, a malicious server could induce the exchange and verification of a large number of SPVs by creating fake server identities and neighbor relationships leading to a DoS attack on the system. Fortunately, simple rate limiting on the number of SPVs accepted by a TN from each neighbor eliminates this risk of a DoS attack.

3.3.4 SNN Maintenance

This section provides additional details on how TrickleDNS accepts new TNs into the system, handles failure and leaving of existing TNs, and revocation or replacement of public keys.

Join: A new TN s joining TrickleDNS needs to know about its neighbors before it can broadcasts its public key. It obtains the set of neighbors by contacting a few existing TNs called *bootstrap servers*. The bootstrap servers return to s their current identity graphs. s constructs its identity graph G_s by applying majority consensus; that is, it accepts a keyed identity if a majority of the bootstrap servers know about that keyed identity. s then identifies its broadcast neighbors from the list of TN identities it has and initiates the broadcast of its keyed identity.

The bootstrap servers are not a centralized pool of servers and neither are they trusted. Any server can serve as a bootstrap server for the joining node. Contacting a large number of bootstrap servers and performing a consensus of their returned state alleviates the risk of being completely misled during the bootstrap process. Alternatively, the joining TN can learn a trusted set of bootstrap servers through out-of-band mechanisms.

Failures and Departures: TrickleDNS employs a heartbeat protocol to detect failures. Each TN s periodically broadcasts a signed *keep alive* including its keyed identity. A TN removes a keyed identity from its identity graph if it fails to receive a few consecutive *keep alives* for that keyed identity.

Key Renewal: Finally, a TN might want to revoke its current public key and start using a new pair of private-public keys. Key revocation and renewal in TrickleDNS is trivial and happens when the TN initiates a SPV broadcast with a new keyed identity with the new public key and a higher sequence number.

Note that the above processes of joins, failures, and key renewals might create temporary inconsistencies in the identity graph; few keyed identities might be for departed or failed nodes or older keys while new keys and node identities may not have been included yet. We intend to tolerate these inconsistencies simply as part of the path disjointedness check where they may cause false positives. A slight increase in the neighbor degree, and

thereby the network connectivity, is sufficient to alleviate the effect of false positives. Moreover, since the TNs are supposed to be infrastructure servers, we expect the churn to be quite low.

3.4 Reliable Name Binding

Reliable Name Binding is a critical step that tightly couples TrickleDNS with the conventional DNS. The end-goal of TrickleDNS is to be a safetynet for the existing DNS as opposed to setting up a completely new namespace. Hence, if a TN X within TrickleDNS claims to be the authoritative nameserver for the domain *foo.bar*, then the ideal property we require from reliable name binding is that: an external client doing a name-lookup through TrickleDNS for *foo.bar* should be redirected to X if indeed X's claim is genuine. If X's claim is not genuine, then TrickleDNS should impede X from even propagating such a claim.

The reliable key distribution mechanism (described in Section **??**) between TNs facilitates the rest of data dissemination within TrickleDNS. Each participating domain D chooses a private-public key pair $(D.k, D.K)$ independently and creates a *domain keyed identity* $D.kid = (D.name, D.K, D.seq)$ for itself. A participating authoritative nameserver s of the domain then broadcast the domain keyed identity on the SNN by signing it with its private key.

To perform reliable name binding, any TN that propagates a domain ownership claim needs to prove to the system that it is indeed one of the authoritative nameservers of the domain. This proof is generated by independent verification by randomly chosen peer TNs called *certifying servers*. Each TN is associated with a *randomly chosen* set of $\mathcal{C}$ peer TNs which act as its certifying servers. The goal is that before a TN can propagate a domain ownership claim within TrickleDNS, it should first prove its domain ownership to its certifying servers.

Each TN s belonging to domain D connects to $\mathcal{C}$ certifying servers in a similar manner as it chooses its broadcast servers. Similar to the concept of "constrained links" within the SNN, these certifying servers are also randomly chosen but explicitly constrained - this is to prevent an adversarial TN from colluding with other TNs to act as its certifying servers. The i^{th} certifying server of s is that server whose identifier is closest to SHA-1$(s.ip_{0\ldots b-1}|i)$ clockwise on the consistent-hashing key space. Here, the different authoritative nameservers of a domain will be mapped to the same certifying servers if their IP addresses are from the same block. This reduces the verification load on the certifying servers, especially when malicious servers request verifications for false claims. Moreover, malicious adversaries only owning a small number of IP address blocks cannot launch DoS attacks by requesting authority certificates.

A certifying server c checks whether a TN s is authoritative for a domain D using the DNS hierarchy or TrickleDNS itself if the parent domain is part of TrickleDNS. c perfoms the check as follows: 1) if c reliably knows the public key of the parent domain already it looks for a cached NS records and glue A records signed by the parent indicating that s is authoritative for D. 2) otherwise, c performs a complete DNS lookup identifying the parent domain's nameservers and fetching the NS and glue A records from them. It then checks that a majority of the parent domain's nameservers acknowledge that s is authoritative for the claimed domain.[2].

The i^{th} certifying server c_i for a TN s provides a signed authority certificate $cert[c_i.id, s.id, D.id]_{c_i.k}$ to s attesting its authority over the domain D, which s broadcasts on the SNN. Any server can verify the certificates using the public key of the certifying server. A server accepts that s is authoritative for D if it has atleast $\lceil \frac{\mathcal{C}}{2} \rceil$ valid authority certificates. Moreover, similar to key distribution, we can bound the number of compromised or malicious servers that name binding can tolerate as follows:

Lemma 1. *Every non-malicious nameserver can be bound to its domain with high probability in the presence of $f * n$ malicious servers where f is an upper bound on the fraction of adversarial nodes provided $\mathcal{C} \geq \beta \log n$ where $\beta = \frac{16(1-f)}{(1-2f)^2} \ln 2$.*

In the above certification process, trust is placed on the current DNS hierarchy. While is clearly a compromise as DNS is not secure in the first place, we believe it's a practical compromise for the following reasons. First, it reduces the large trusted computing base (TCB) involved in peer-to-peer DNS alternatives to the much smaller TCB along the DNS hierarchy. Second, higher-level, parent domains typically tend to have better redundancy than low-level domains as will be clear from Section **??**. Finally, in the absence of DNSSEC or an alternative central certification authority, an approach based on independent verification and consensus provides some resilience against compromises in current DNS.

3.5 Pushing DNS Records

TrickleDNS proactively propagates only a certain set of critical records that include the delegation or NS records used to identify the authoritative namservers of a domain, the corresponding glue (A) records that provide the IP

[2]Looking for agreement might generate false positives because sometimes DNS servers respond with different set of records depending on the location of the server or the client. However, this is not a serious problem for NS records which are seldom generated dynamically.

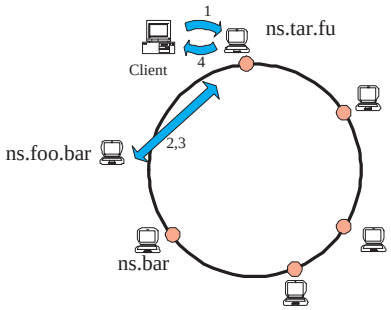

Figure 1: **A lookup on a domain that is not part of TrickleDNS. Here $foo.bar$ is outside TrickleDNS while bar is part of TrickleDNS. The local DNS server contacts $ns.bar$ which redirects it to $ns.foo.bar$.**

address of the nameservers, and the start of authority or SOA records. Consequently, a TN typically needs to perform a single lookup to resolve a DNS query for a participating domain (a domain redirection through a CNAME response, however, might require additional lookups). We could avoid even this lookup by proactively disseminating every record associated with a domain. However, this considerably increases the bandwidth overhead on the TNs diminishing their incentive to participate. Alternatively, we could enable proactive dissemination for popular records (mail servers, web servers, etc.). Doing so requires additional mechanisms to ensure that participating domains do not overload the system.

A client can resolve a name, say *foo.bar*, using TrickleDNS as follows: First, a client's DNS servers are set to point to one or more trusted nameserver in the TrickleDNS network. These may be the nameservers of the client's local domain itself or other open-access TrickleDNS nameservers. A TrickleDNS namesever then handles the query for *foo.bar* in the following way: If foo.bar is a participating domain, then the nameserver uses its locally cached records for the domain's authoritative nameservers and queries them. If foo.bar is not a participating domain, then the nameserver executes a regular DNS recursive resolution process starting with the authoritative nameserver of the immediate participating parent domain. However, the TN obtains a majority consenses for the returned records whenever possible by querying multiple nameservers during each recursion. Figure 1 illustrates the resolution of a DNS query in TrickleDNS.

4 Security of TrickleDNS

We already outlined how TrickleDNS deals with certain critical attacks in Section 3. Here, we give a top-down summary of the security properties of TrickleDNS and discuss their implications.

A fully-decentralized, peer-to-peer solution to DNS such as TrickleDNS faces at least three key types of attacks from malicious adversaries:

1. **Server Compromises:** An adversary might take advantage of a software vulnerability and compromise one or more participating nameservers. While TrickleDNS cannot eliminate software vulnerabilities or prevent compromises, it makes a successful domain hijack from compromised TrickleDNS servers incredibly difficult. In order to successfully hijack a domain, the adversary either needs to compromise a majority of the certification servers, called a *certification attack*, or a sufficient number of servers in the paths between the domain's authoritative nameservers and other TNs, called a *path attack*.

   TrickleDNS achieves this high resilience against certification and path attacks through randomization, that is, distributing the vulnerable servers uniformly in the overlay. The success of certification attack depends on the probability of finding a majority vulnerable servers among certification servers, a value that can be made very low by increasing $\mathcal{C}$ as required. Similarly, the probability of success of a path attack can be made as low as desired by increasing the neighbor degree $\mathcal{B}$ as required.

   Of course, an adversary can also hijack a domain by compromising one or more of its parent domain nameservers (*parental attack*). If some of the parent domains are part of TrickleDNS, then the above security analysis also applies to them as well. Otherwise, this dependence of current DNS hierarchy is an unavoidable risk. We hope that looking for a majority consensus between the parental servers would alleviate this risk to some extent.

2. **Identity Attacks:**

   An adversary might be able to increase its chances of succeeding in a path attack or certification attack by artificially increasing the number of malicious servers in the system or breaking randomization by controlling the identity of the server. One method of identity attack accessible to most domain owners is to take advantage of the IP address space the domain owns and introduce as many servers as possible from that address block or carefully pick an IP address for its server so as to place it strategically in the overlay. TrickleDNS overcomes this attack by using a short prefix of the IP address during neighbor selection.

   However, a rich and powerful adversary might be able to launch an identity attack through other means. For instance, it could use a large network of compromised hosts called a Botnet to launch the attack from widely different IP addresses. While TrickleDNS is very vulnerable to a Botnet attack, a few precautionary measures can reduce the risk of such attacks and make it easier to detect them: First,

verify that the host has service on port 53 (the DNS port); often times, Bots are behind a firewall and may not be able to receive traffic on port 53. Second, ensure that the host has a valid reverse DNS (PTR) record; Bots typically have dynamic (DHCP) IP addresses with no individual reverse DNS records. Finally, use publicly-available black-lists to reject hosts known for other Bot activities.

Another, more subtler way to launch an identity attack is to attack the IP layer by a) spoofing IP addresses, b) IP hijacking, or c) man-in-the-middle attacks. TrickleDNS is immune to IP spoofing because it performs two-way communication using TCP. IP hijacking by compromising Internet routing, however, could be dangerous; if the hijack is partial it is likely to have less impact as TrickleDNS connects each server to several others randomly distributed in the Internet. On the other hand, a complete hijack can be treated as a compromised IP address or IP address block; the above analysis for server compromises holds for IP hijacks as well. Man-in-the-middle attacks have a similar impact as hijacked IP addresses and can be treated as a server compromise.

3. **Denial of Service Attacks:**

   Finally, adversaries can try to disrupt TrikcleDNS by launching general DoS attacks on the general system or a more targeted DoS attack on selected domains. Since proactive propagation of a domain D's public key ensures that any TN can serve D's signed records, a general DoS attack on D's authoritative nameservers does not disrupt service for that domain. Instead, the attacker can try to disrupt key distribution by attacking TNs. In this case, randomization, which alleviates server compromises, also diminishes the success of DoS attacks. Moreover, all heavy-weight operations such as signature and authority verification in TrickleDNS are rate-limited to avoid DoS attacks.

Finally, we expect that a participating domain interested in its own security will take the necessary measures (for instance, apply patches) to secure its own nameservers. TrickledDNS does not protect a domain from compromises to its own nameservers. Its goal instead is to protect a participating domain from vulnerabilities in other, less-secure domains.

Dynamically generated DNS records (for example, DHCP addresses) require the private key to be stored in memory to sign records online posing a risk of key compromise. This problem can be mitigated by isolating the signing process and running it on a node that is protected within a firewall, only communicates on restricted ports, and does not also run the name server.

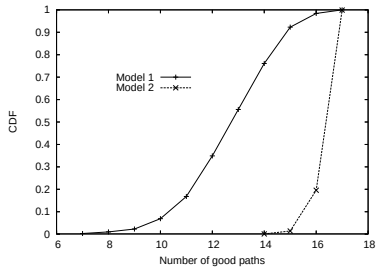

Figure 2: **CDF of good paths for 65536 servers and malicious fraction = 5 % in adversary model 1.**

XXX Add the above to previous section.

5 Evaluation

In this section, we evaluate the security properties and the performance of TrickleDNS.

5.1 Security Analysis

In this section, we evaluate the path, certification, and parental resilience of TrickleDNS. To evaluate the first two properties, we randomly generated SNN topologies and analyzed the connectivity of these topologies. To evaluate the third , we analyzed the parental relationships in current DNS.

5.1.1 Path Resilience

We generated topologies in which participating servers establish links to their broadcast servers. The neighbor degree $\mathcal{B}$ was set to $\log n$ for a network of size n. We define a *good path* in the generated topology as a path from source to destination passing only through non-malicious nameservers. For reliable communication between a pair of servers, the number of good paths must be atleast $\lceil \frac{1}{2} \log n \rceil$. We call such a pair of servers a reliably communicating pair. We check whether this condition holds for pairs of non-malicious servers in the overlay.

We consider two models of adversarial behavior. In *Model 1*, the adversary controls a set of IP addresses distributed uniformly at random over the IP address space. In *Model 2*, the adversary controls a limited number of IP prefixes. We further assume that all the adversarial entities may collude to cause maximum damage to the system and that these entities may behave in a Byzantine manner. In a practical setting, an adversry may also launch a DoS attack on the servers. We do not explicitly model these in the evaluation here since the adversary models that we consider are more powerful. Further, these simulations also indicate the impact of benign server failures.

We now present the results for each of the adversary models.

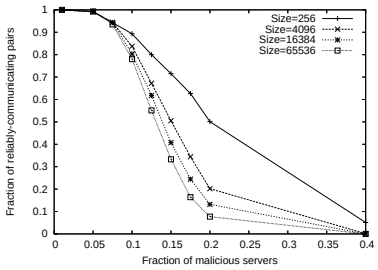

Figure 3: **Reliably communicating pairs for different values of malicious fractions in model 1.**

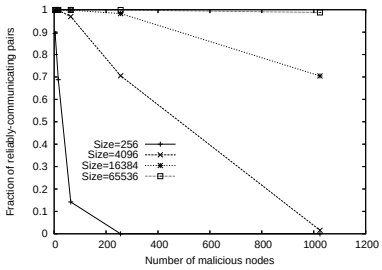

Figure 4: **Reliably communicating pairs for different number of malicious entities in model 2.**

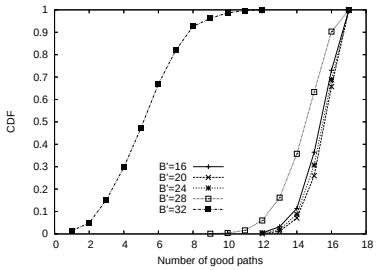

Figure 5: **CDF of good paths for different B' in a network of size 65536 under adversary model 2.**

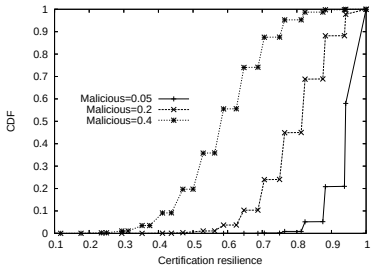

Figure 6: **CDF of the fraction of non-malicious servers in the certifying server set in adversary model 1.**

Model 1: In this model, the adversary controls a set of IP addresses distributed uniformly at random over the IP address space. We quantify the maliciousness in terms of the fraction f of malicious servers in the system. Figure 2 shows the CDF of the number of good paths between server pairs for a system with 65536 servers of which 5% are malicious. More than 95% of server pairs have atleast 9 paths and can surely communicate reliably. This is still an underestimate of the number of reliably communicating pairs. In fact, only 0.01% of the pairs cannot communicate reliably since in most of the cases the number of bad paths is small as well (In this setting, we expect each server to have only 1.2 malicious neighbors).

Figure 3 shows the fraction of reliably communicating pairs as the maliciousness increases. These numbers are an average of 10 runs of 1000 lookups each. We see that a system with 65536 servers can tolerate 5% malicious fraction. For large systems, an increase in the malicious fraction causes a steeper decline in the fraction of reliably communicating pairs due to the greater number of hops traversed by each SPV.

XXX What's the point of the prev para?

Model 2: In this model, the adversary controls entire 24s. We quantify maliciousness here as the number of prefixes owned by each adversary. Figure 2 shows the CDF of the good paths for a system with 65536 servers and 64 malicious entities each spoofing an entire 24. The length of IP prefix used in the secondary identifier is also 24. By choosing neighbors based on the 24, the influence of the malicious servers is reduced to the number of 24's owned. This is reflected well in figure 2, where there are atleast 12 reliable paths between server pairs.

Figure 4 shows the effect of system size vs. adversary size on the fraction of reliably communicating pairs. The small systems (256 and 4096) are overwhelmed by the malicious servers once the number of malicious servers approaches the number of non-malicious servers. For large systems, the probability of reliable communication remains at 1 even when there are 1024 malicious entities. This is due to the prefix-based neighbor selection method used. Figure 5 illustrates this for a system of size 65536 having 256 malicious entities. The graph $B' = 32$ is equivalent to picking neighbors without any prefix clustering. We see that this scheme performs badly with about 90% of the pairs having less than 8 good paths. However, when the prefix length B' is set to 24, all pairs can communicate reliably. Further increases in the prefix length B' do not seem to improve the probability of reliable communication.

5.1.2 Certification resilience

To measure certification resilience, we generated random topologies as in Section 5.1.1 in which each server set up links to $\log n$ certifying servers. Figure 6 shows that all servers have less than 20% malicious servers in their certifying server set even with 1024 malicious entities each controlling a 24. Figure 7 shows that even when 20% of the servers are malicious, only about 1% of the servers will not be able to obtain a certificate of authority. Thus the certification resilience of the system is high.

XXX: confused. which is model 1 and which model 2 above?

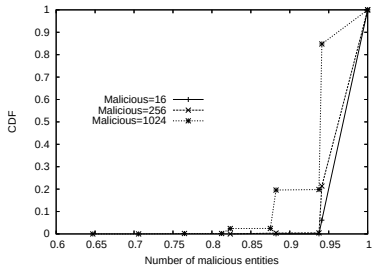

Figure 7: **CDF of the fraction of non-malicious servers in the certifying server set in adversary model 2.**

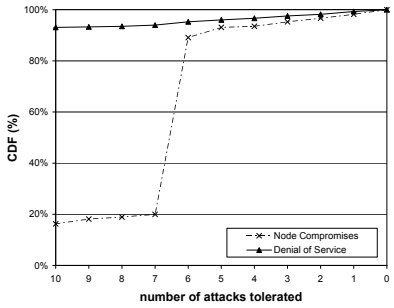

Figure 8: **CDF of the number of node compromises and DoS attacks that can be tolerated by existing domains.**

5.1.3 Parental Resilience

To measure parental resilience, we used a snapshost of inter-domain relationships in DNS as it existed on July 22, 2004. This snapshot was generated from a study done at Cornell University [27] and contains $166,771$ distinct name servers that contribute towards resolution of $597,196$ distinct domains.

It is important to analyze the resilience in the parental path as limited parental resilience undermines the reliable binding of a nameserver to its domain. Fortunately, Figure 8 shows that resilience in DNS is higher for the parent domain with close to 90% of the parent domains able to tolerate atleast 6 node compromises. In contrast, more than XXX% of domains have only two nameservers indicating that a compromise of one of them could be fatal. The good parental resilience is expected because DNS hierarchy is flat and a large number of domains fall directly under the top-level domains. In Figure 8, the sharp increase in parental resilience at 6 corresponds to the large number of *.com* domains that are served by the thirteen *.com* name servers.[3] Figure 8 also shows that resistance to DoS attacks for parent domains is good.

The analysis in this section tells us that i) the reliable communication mechanism and the certification mechanism can tolerate a small yet significant malicious presence, ii) that using the parental nameservers to verify the authority of a nameserver before it joins TrickleDNS is a reasonable approach. Further, these properties have been achieved with $\log n$ broadcast and certifying servers. These system parameters can be increased by a constant factor with little added overhead so that the path and certification resilience can be improved. On the other hand, the parental resilience is not easy to improve.

[3]In reality, there are more than thirteen *.com* nameservers behind the thirteen published IP addresses. In this analysis, we just count parents by the number of distinct NS records returned.

5.2 Performance Analysis

In this section, we describe experiments to measure the performance of TrickleDNS and compare it with legacy DNS. Here, we are interested in quantifying the following three metrics: 1) lookup latency of DNS queries in TrickleDNS , 2) time taken to push DNS records between name servers, and 3) memory and bandwidth overhead of pushing DNS records.

For benchmarking performance, we implemented the TN functionality based on the djbdns [6] codebase. The mechanisms for reliable communication described in Section **??** are implemented as a reliable communication toolkit and exported to the TrickleDNS implementation codebase. Our evaluation used 1024-bit RSA keys for authentication. The TN implementation is layered on top of the toolkit. Each TN acts as a caching DNS nameserver that can support the operations and optimizations present in current DNS.

We deployed our implementation of the TN on the 62-node PSI cluster [5]. To compare the lookup performance of TrickleDNS with legacy DNS, each TN acts as an authoritative nameserver. Queries are sent to a randomly chosen TN. The TN tries to answer the query from its cache; otherwise it queries legacy DNS and reflects the response record. Whenever a TN gets a new record, it pushes it to the other servers. This gives us an estimate of the overhead of pushing DNS records. Queries are generated from a real workload collected by Jung *et al.* [19] at MIT.

XXX Need more workload details. Number of queries, domains, etc.

In our testbed, we executed 60 TN instances on different nodes with each instance having an average degree of 6. Each TN instance reliably discovered the other instances in the network and then pushes nameserver records to the these instances.

5.2.1 Lookup Characteristics

We first compare the lookup latencies in TrickleDNS and the legacy DNS. From the discussion on the lookup process in Section 3.5, we notice that the time to perform a lookup is dependent on whether the client's local DNS server is part of the TrickleDNS. Given a target domain D, if the local DNS server is part of the TrickleDNS, it can respond to a query with the NS and glue records

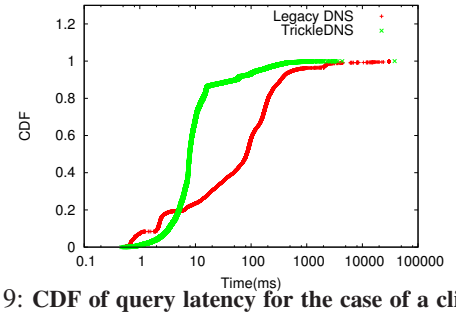

Figure 9: **CDF of query latency for the case of a client directly contacting an SENs compared with the legacy DNS latency.**

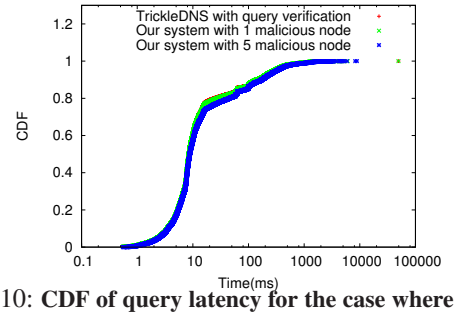

Figure 10: **CDF of query latency for the case where the local DNS server that is not an SEN.**

of an authoritative name server of D. Otherwise, the local server reflects the query to a TN. To enhance security, the local server may simultaneously queries different TNs and wait for a majority consensus.

The first experiment deals with the scenario where the local DNS server is a TN. In this case, Figure 9 shows the distribution of query latency for TrickleDNS and legacy DNS. Only queries which were not answered from the local cache were included in the measurement. We measure the latencies of those queries for which the TNs are already populated with the target records. Otherwise, the query latency would include the latency of fetching the record via legacy DNS. The Figure shows that the median latency of TrickleDNS is a factor of 10 lower than legacy DNS. Note that the latencies shown in both cases are simply the latencies for fetching the authoritative NA and glue A record. The DNS clients then have to contact the authoritative server for the target A record which will incur an additional delay irrespective of whether legacy DNS or TrickleDNS is used.

In the second experiment, we consider the case in which the client's local DNS server is not part of TrickleDNS . The local server then reflects the query to a set of TNs and takes a majority vote on their responses. In our experiments, the resolver contacts 3 TNs for redundancy. The latencies for this experiment is shown in Figure 10.

We consider three scenarios corresponding to no node in the network being malicious, and 1 and 5 nodes being malicious respectively. A malicious node when contacted simply allows the query to time out. The first observation is that, in the median case, queries take almost the same time as the case where only one node is contacted. Further, the addition of a single malicious node does not affect the latencies because the servers are chosen uniformly at random for each query. With 5 malicious nodes, the median latency only increases by about 8%. With a larger network size, a single malicious node will have a smaller influence on the query latency.

To summarize, for domains that are part of TrickleDNS , queries are answered much faster than legacy DNS both in the cases where the local DNS server is a TN and when it is not (and must send out redundant queries). Although our evaluation tries to simulate the effect of queries in a real TrickleDNS deployment without doing any form of namespace partitioning, the real-world performance of such a system would depend on the regions of push-pull, the nature of the resolver, and the latency in fetching the final A record.

5.2.2 *Overhead of Pushing DNS records*

The 90^{th} percentile of the update time for reliable communication and for broadcasting DNS records shows the latter to be 25 times slower than the former (4.5 seconds versus 180 ms). This is due to the path-vector signature operations required in reliable communicaiton. However, reliable communication is performed infrequently (on the order of days) since the underlying topology is not very dynamic. Once reliable communication between the TNs, subsequent propagation and verification of DNS records incurs very low overhead.

XXX Where's the ref fig for above para. If no fig is refered, then the first sentence needs to be rephrased.

The system profiles for the TN instances indicate that TrickleDNS incurs low bandwidth and memory overhead. The bandwidth exchanged between TNs is 1.35 KBps of which only 12% of the bytes are used in reliable communication. This represents a very small bandwidth overhead. The net memory usage of each TN was roughly 9.356 MB of memory at the end of the trace with 6804 records being stored. The state maintained and propagated per domain by our implementation is less than 10 KB which is a modest quantity.

Recent work by Handley *et al.* [18] shows that roughly 0.5% of domains change name servers and about 0.1% of domains expire every day. Extrapolating to the entire DNS, they claim that roughly $420,000$ domains change nameservers and $100,000$ domains expire everyday. For the entire DNS this translates to an update rate of 1.6 Kbps [18], a rate that can very easily be handled by our system.

From a cryptographic verification standpoint, our implementation can verify the validity of roughly $40,000$

signatures every second. Hence, the verification overhead associated with proactive dissemination is negligible. Additionally, offline verification ensures that this verification process does not affect the performance of DNS lookup.

In summary, the overhead of pushing name server records is reasonably small in terms of the amount of state maintained, bandwidth requirements, memory requirement and processing overhead.

6 Limitations

A decentralized design that does not rely on any form centralized trust suffers inherent drawbacks. The foremost is the inability to provide absolute data integrity. TrickleDNS tolerates a certain number of adversaries in the system depending on the connectivity of the reliable communication network, number of authoritative nameservers, and security enforcement in the legacy DNS parental path. The security of the first can be increased by forming a denser network with a trade-off between communication overhead and increased resilience. The second requires effort from the concerned domain to improve the number and security of its name servers. The third is outside the control of TrickleDNS and the concerned domain. Nevertheless, our analysis described in Section 5.1.3 shows that domains in the top of the DNS hierarchy typically have better resilience than lower-level domains.

A particularly difficult attack for TrickleDNS to tolerate is a Sybil attack generated from Botnets, where the servers have wide-ranging IP addresses and do not fit into our model of Sybil attackers. However, since participating servers are first verified to be authoritative name servers of some domain, the scale of the attack can be mitigated. However, note that, adversaries still have the freedom of setting any IP address to the servers they introduce, and can use this freedom to put malicious servers at strategic points in the SNN. The restriction on IP address prefixes helps to mitigate precisely this problem.

Other potential security issue involves DNS dynamic updates, cache poisoning, and IP spoofing. Dynamically generated DNS records (for example, DHCP addresses) require the private key to be stored in memory to sign records online posing a risk of key compromise. This problem can be mitigated by isolating the signing process and running it on a node that is protected within a firewall, only communicates on restricted ports, and does not also run the name server. TrickleDNS is implemented using DJBDNS, which is resilient to cache poisoning. IP spoofing does not affect reliable communication since servers use two-way communication based on TCP.

7 Conclusions

This paper argues for a decentralized approach for securing DNS. Despite the prevalence of DNSSEC for more than ten years, it is yet to gain acceptance. TrickleDNS can provide an incremental deployment path by enabling domains to independently improve their resilience against DNS security threats such as domain hijacks and DoS attacks. Proactive dissemination of DNS records on a secure network further improves lookup and update performance.

References

[1] Crackers cripple RSA Server. `http://www.computeruser.com/newstoday/00/02/15/news2.html/`.

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the Domain Name System Security Extensions. Request for Comments 4035, March 2005.

[3] G. Ateniese and S. Mangard. A new approach to dns security (dnssec). In *ACM CCS '01*, pages 86–95, New York, NY, USA, 2001. ACM Press.

[4] D. Atkins and R. Austein. Threat Analysis of the Domain Name System. Request for Comments 3833, August 2004.

[5] Berkeley Millennium Cluster. http://www.millennium.berkeley.edu/PSI/index.html.

[6] D. Bernstein. djbdns. `http://cr.yp.to/djbdns/notes.html`.

[7] BIND Vulnerabilities. http://www.isc.org/sw/bind/bind-security.php, February 2004.

[8] N. Brownlee, kc claffy, and E. Nemeth. DNS Measurements at a Root Server. In *Proc. of IEEE GlobeCom*, San Antonio, TX, November 2001.

[9] N. Brownlee, kc Claffy, and E. Nemeth. DNS Root/gTLD Performance Measurements. In *Proc. of Usenix Systems Administration Conference*, San Diego, CA, December 2001.

[10] C. Cachin and A. Samar. Secure distributed dns. In *DSN*, pages 423–432, 2004.

[11] R. Cox, A. Mutitacharoen, and R. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *Proc. of IPTPS*, Cambridge, MA, March 2002.

[12] T. Deegan, J. Crowcroft, and A. Warfield. The main name system: an exercise in centralized computing. *SIGCOMM Comput. Commun. Rev.*, 35(5):5–14, 2005.

[13] DNSSEC. http://www.dnssec.net.

[14] J. Douceur. The sybil attack. In *Proc. of IPTPS*, 2002.

[15] D. Eastlake. Domain Name System Security Extensions. Request for Comments 2335, March 1999.

[16] C. Fetzer, G. Pfeifer, and T. Jim. Enhancing dns security using the ssl trust infrastructure. In *Proceeding of the IEEE WORDS 2005*, 2005.

[17] M. J. Freedman, E. Sit, J. Cates, and R. Morris. Introducing tarzan, a peer-to-peer anonymizing network layer. In *Proceedings of the IPTPS02)*, Cambridge, MA, March 2002.

[18] M. Handley and A. Greenhalgh. The Case for Pushing DNS. In *Proc. of HotNets*, November 2005.

[19] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. Dns performance and the effectiveness of caching. In *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 153–167, New York, NY, USA, 2001. ACM Press.

[20] J. Kangasharju and K. W. Ross. A Replicated Architecture for the Domain Name System. In *Proc. of IEEE INFOCOM*, 2000.

[21] M. Lad, A. Nanavati, D. Massey, and L. Zhang. An algorithmic approach to identifying link failures, 2003. `http://www.nge.isi.edu/~masseyd/pubs/massey_prdc04.pdf`.

[22] P. Mockapetris. Domain Names: Concepts and Facilities. Request for Comments 1034, November 1987.

[23] P. Mockapetris. Domain Names: Implementation and Specification. Request for Comments 1035, November 1987.

[24] V. Pappas, Z. Xu, S. Lu, D. Massey, A. Terzis, and L. Zhang. Impact of Configuration Errors on DNS Robustness. In *Proc. of ACM SIGCOMM*, Portland, OR, August 2004.

[25] L. Poole and V. S. Pai. ConfiDNS: Leveraging scale and history to improve DNS security. In *Proceedings of WORLDS 2006*, Seattle, WA, November 2006.

[26] V. Ramasubramanian and E. G. Sirer. The Design and Implementation of a Next Generation Name Service for the Internet. In *Proc. of ACM SIGCOMM*, Portland, OR, August 2004.

[27] V. Ramasubramanian and E. G. Sirer. Perils of Transitive Trust in the Domain Name System. In *Proc. of ACM IMC*, Berkeley, CA, October 2005.

[28] A. Rowstorn and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, November 2001.

[29] C. Schuba. Addressing Weaknesses in the Domain Name System Protocol, August 1993. Masters Thesis, Purdue University.

[30] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, San Diego, CA, August 2001.

[31] M. Theimer and M. B. Jones. Overlook: Scalable name service on an overlay network. In *ICDCS '02*, page 52, Washington, DC, USA, 2002. IEEE Computer Society.

[32] X. Wang, Y. Huang, Y. Desmedt, and D. Rine. "enabling secure on-line dns dynamic update". In *16th Annual Computer Security Applications Conference (ACSAC'00)*, page 52, 2000.

[33] Z. Yang. Using a byzantine fault tolerant algorithm to provide a secure dns, 1999.