On the Performance and Analysis of DNS Security Extensions *

Reza Curtmola Aniello Del Sorbo Giuseppe Ateniese
Information Security Institute and
Department of Computer Science
Johns Hopkins University, Baltimore, MD 21218, USA
E-mail: {crix, anidel, ateniese}@cs.jhu.edu

Abstract

The Domain Name System (DNS) is an essential component of the critical infrastructure of the Internet. The role of DNS is vital, as it is involved in virtually every Internet transaction. It is sometimes remarked that DNS works well as it is now and any changes to it may disrupt its functionality and add complexity. However, due to its importance, an insecure DNS is unacceptable for current and future networks. The astonishing simplicity of mounting an attack against the DNS and the damaging potential of such an attack should convince practitioners and system administrators to employ a secure version of DNS. However, security comes with a cost. In this paper, we examine the performance of two proposals for secure DNS and we discuss the advantages and disadvantages of both. In particular, we analyze the impact that security measures have on the performance of DNS. While it is clear that adding security will lower DNS performance, our results show that the impact of security can be mitigated by deploying different security extensions at different levels in the DNS tree.

We also describe the first implementation of the SK-DNSSEC [2] protocol. The code is released under an open-source license and is freely available at `http://skdnssec.isi.jhu.edu`.

1 Introduction

The Domain Name System (DNS) is one of the world's largest distributed databases, whose main function is to translate human readable *domain names* to their corresponding IP *addresses*. Its tree-like structure allows a hierarchical distribution of domain names that facilitates fast name resolution and sub-division of the management load for domain administrators. The role of DNS is vital as it is involved in virtually every Internet transaction. Considering the importance of DNS, it is surprising that a secure version of it is not currently deployed. Vulnerabilities in the DNS system were noticed as early as 1990, in the seminal paper by Bellovin [3]. Several known threats to the DNS system are summarized in [4], some of which include packet interception, packet ID guessing, query prediction and cache poisoning. Because the DNS packets are not cryptographically signed, it is possible for a malicious party to inject, intercept or modify these packets with the intent of disrupting the DNS service [3, 4, 5, 6].

In order to have a secure DNS, two security requirements have to be met at a minimum: *Data origin authentication* and *data integrity*. The main proposal to extend the existing DNS to make it secure is based mostly on public-key cryptography (PK-DNSSEC [7]). This proposal has received a lot of attention and exists as an IETF standard. A different solution, proposed in [2] (SK-DNSSEC), makes use almost exclusively of symmetric-key cryptography.

This work presents the first implementation of the SK-DNSSEC protocol. A functional implementation of SK-DNSSEC allowed us to compare its performance with plain-DNS and PK-DNSSEC. We can thus evaluate the performance tradeoff induced by the security overhead and identify the advantages and disadvantages of both security extensions. We collected a real DNS trace and used it to model the traffic pattern and zone contents in our experiments. With regard to the computational cost for a name server, our results show that PK-DNSSEC outperforms SK-DNSSEC for authoritative and referral name servers, while SK-DNSSEC exhibits better performance for recursive name servers. We argue that a hybrid approach with PK-DNSSEC deployed for top-level domains, where the information is static, and SK-DNSSEC for low-

*This is the full version of the paper that appears in the proceedings of CANS 2005 [1]

level domains, where the information is more dynamic, would leverage the benefits of both worlds.

Our experiments also show that PK-DNSSEC generates considerably more network traffic and has higher query latency than plain-DNS or SK-DNSSEC. Furthermore, SK-DNSSEC exhibits several other advantages over PK-DNSSEC, some of which are: it has simpler key management, it is less intrusive for zone files and it uses less memory for caching. All these aspects make SK-DNSSEC a valid alternative to PK-DNSSEC, especially if DNS security is needed in dynamic environments.

The rest of this paper is structured as follows. We review background and related work in Section 2. We present some details of the SK-DNSSEC implementation in Section 3. In Section 4 we empirically evaluate the performance of plain-DNS, SK-DNSSEC and PK-DNSSEC and conduct a comparative analysis of these three models. In Section 5 we discuss several aspects that can have a significant impact on the functionality of a secure DNS. Section 6 concludes the paper.

2 Background and Related Work

A *zone* is a part of the domain name space and the name server that manages a zone is called *authoritative* for that zone. The basic data unit in a zone is a *Resource Record (RR)*. Clients that query name servers are called *resolvers*. The process by which resolvers retrieve data on a domain name is called *resolution*, and it usually involves a series of queries to servers along the path from the root node to the target name. A *recursive (caching)* name server, upon receiving a query, will resolve the query, cache it and return the answer. A *referral* name server does not return a final answer, but rather does a referral, meaning it redirects the query to the next name server in the DNS tree on the path to the server authoritative for the queried name.

The Public Key DNS Security Extensions. PK-DNSSEC uses three new Resource Records (*RR*) in order to provide end-to-end authenticity and data integrity: `KEY` (to encode the public key associated with a zone), `SIG` (to encode digital signatures over an *RR* set) and `NXT` (to indicate what does not exist in a zone). DNS servers are required to sign the *RR* sets in the zones for which they are authoritative, and answer queries by returning the corresponding `SIG RR`s along with the queried resource record set. An authenticated `NXT RR` is returned to indicate that a queried *RR* does not exist in the zone. On the other hand, a DNS resolver is required to verify signed answers by validating the `SIG RR`s that cover each *RR* set. The resolver can be configured to trust a set of public keys that correspond to a set of zones. If the answer is from a zone whose public key is trusted, the resolver can perform the verification without taking additional steps. Otherwise, the resolver needs to establish a chain of trust starting from one of the trusted public keys (usually of the root name server) down to the public key of that zone. During this process, the resolver may need to make additional queries for public keys of intermediate name servers.

The SK-DNSSEC protocol. PK-DNSSEC is based on public-key cryptography and places a considerable computational burden on resolvers as they have to verify the authenticated DNS answers. Moreover, the answers containing signed *RR* sets generate considerably more network traffic than plain DNS. In an effort to minimize such undesired effects, SK-DNSSEC [2] proposes a different approach, mostly based on symmetric key algorithms. SK-DNSSEC introduces the notion of DNS symmetric certificates which provide integrity and authenticity by combining encryption techniques with MAC functions (specifically HMAC [8, 9]). A DNS symmetric certificate is similar to a public-key certificate in the sense that it binds the owner's identity to a key. To obtain a secure answer, a DNS resolver establishes a chain of authentication from a trusted DNS server to the authoritative name server using symmetric certificates. Initially, the resolver needs to acquire a long-term root certificate from a root server. This is the only step in which public-key cryptography is used, and it is done only once in order to bootstrap the chain of trust. Root certificates are never queried again until they expire, usually when the public key of the root server changes. Each node in the DNS hierarchy shares a symmetric key with its parent, called *master key*. The root does not share its master key with any other node. Master keys are used to generate symmetric certificates which allow safe transport of secret keys from the parent to the child in the DNS tree. A resolver needs to acquire a DNS symmetric certificate for each DNS server encountered while the chain of trust is being built from the root server to the authoritative name server. These certificates can be cached and contain the secrets shared by the resolver with the DNS servers queried during the resolving process. Thus, a DNS server does not need to store any of the information shared with the resolvers.

We illustrate the operation of SK-DNSSEC with an example. We use the notation defined in Table 1. Assume a resolver U wants to get the address of `host.example.com`. If U does not have a root certificate yet, it asks the root for one. The root R_0 generates

K_{XY}	secret key pair (K^1_{XY}, K^2_{XY}) shared by X and Y (Y's master key)
K^1_{XY}	secret key shared by X and Y used for encryption
K^2_{XY}	secret key shared by X and Y used for MAC functions
K_{R_0}	root's (R_0) key pair $(K^1_{R_0}, K^2_{R_0})$ (root's master key)
P_{XY}	symmetric certificate shared by X and Y
$Info(P_{XY})$	relevant information about P_{XY} (such as the identities of X and Y, inception and expiration dates etc)

Table 1: Notation

a key K_{R_0U} which will be shared between the root and the resolver, and which allows the establishment of a private and authentic channel between the root and the resolver in the future. A root certificate P_{R_0U} contains an encryption of K_{R_0U}, under the root's master key K_{R_0}:

$$P_{R_0U} = Info(P_{R_0U}, E_{K^1_{R_0}}(K_{R_0U}, \text{MAC}_{K^2_{R_0}}(Info(P_{R_0U}, K_{R_0U}))$$

Using the root certificate, U queries the root server, which will perform a referral to R_1, the authoritative server for `.com`. In the referral process, the root server generates a key K_{R_1U}, which will be shared by U and the server R_1. The root sends K_{R_1U} to U encrypted under K_{R_0U}, along with a symmetric certificate P_{R_1U}. The symmetric certificate contains an encryption of the key K_{R_1U} under $K_{R_0R_1}$ (the master key of R_1):

$$P_{R_1U} = Info(P_{R_1U}, E_{K^1_{R_0R_1}}(K_{R_1U}, \text{MAC}_{K^2_{R_0R_1}}(Info(P_{R_1U}, K_{R_1U}))$$

U then queries R_1 (authoritative for `.com`) by sending the original DNS request and the certificate P_{R_1U}. The server R_1 retrieves K_{R_1U} from P_{R_1U} and generates a new key K_{R_2U}, which will be shared by U and R_2, the server authoritative for `example.com`. Next, R_1 refers the resolver U to R_2 by creating a new symmetric certificate P_{R_2U}, and sending it along with an encryption of K_{R_2U} under K_{R_1U}. Similarly with P_{R_1U}, the certificate P_{R_2U} contains an encryption of the key K_{R_2U} under the master key of R_2:

$$P_{R_2U} = Info(P_{R_2U}, E_{K^1_{R_1R_2}}(K_{R_2U}, \text{MAC}_{K^2_{R_1R_2}}(Info(P_{R_2U}, K_{R_2U}))$$

Finally, when R_2 is contacted by U with P_{R_2U}, it can retrieve K_{R_2U} from P_{R_2U}. Since R_2 is authoritative for `.example.com`, it can send to U the IP address of `host.example.com`, authenticated with K_{R_2U}.

The strategy deployed in SK-DNSSEC is similar to the one introduced by Davis and Swick [10] and Kerberos [11, 12]. In particular, DNS symmetric certificates can be viewed as *tickets* used by the ticket-granting server in Kerberos to provide clients with access capabilities to certain resources. We refer the reader to [2] for a more detailed description of SK-DNSSEC and its operation.

Other solutions. A P2P-based solution has been proposed in [13], which offers to improve DNS performance and availability. However, the security model in this solution has not been fully developed.

3 Implementation

One of the major contributions of this work is the implementation of the SK-DNSSEC protocol. We emphasize that ours is the first public release of an implementation since the SK-DNSSEC protocol was originally proposed in [2]. Depending on the community's response to this implementation, there are plans to contact IETF for standardization.

BIND is the most widely deployed implementation of DNS protocols and its source code is freely available. We selected version 9 of BIND as the base for the SK-DNSSEC implementation. This version provides the most complete implementation of the PK-DNSSEC extensions.

The current version of the SK-DNSSEC implementation uses AES in CBC mode as the symmetric cipher, HMAC with MD5 [1] as the MAC function, and RSA with PKCS1 padding as the public-key cipher. The implementation can be easily extended to accommodate additional algorithms. However, some algorithms may not be appropriate. For example, Blowfish is a symmetric cipher with a high speed encryption rate, but in our experiments it did not perform as expected. This is due to the fact that Blowfish has a low key agility as opposed to other standard algorithms [16]. In particular, switching frequently between different keys, as required by SK-DNSSEC, can significantly influence the throughput. Thus, we recommend the use of symmetric ciphers with high key agility, like AES.

A name server running BIND contains two major entities: a server component (handling incoming re-

[1] Even if MD5 is not collision resistant ([14, 15]), HMAC-MD5 still provides adequate security in the context of this application.

quests and outgoing responses) and a resolver component (handling outgoing requests and incoming responses). Usually, these two entities interact with each other as part of different name servers. The implementation of the SK-DNSSEC protocol has been adapted to the structure of a DNS server. Each of the server and resolver components can be further split into a receiving and a sending part. Thus, the completion of a query-answer cycle can be seen as a loop of four steps. To enable the SK-DNSSEC extensions, BIND was modified in these four locations: *outgoing query processing* (when the resolver sends the query), *incoming query processing* (when the server receives the query), *outgoing answer processing* (when the server replies with the answer) and *incoming answer processing* (when the resolver receives the answer).

PK-DNSSEC authenticates DNS messages by including new types of resource records in the message, such as public-key signature (`SIG`) *RR*s. Instead, we have chosen to let SK-DNSSEC handle DNS messages as opaque data and authenticate them by appending binary data at the end of the message. Non SK-DNSSEC-aware DNS servers can simply ignore (or log the existence of) this additional data, thus backward compatibility is provided.

Due to space constraints, we describe the SK-DNSSEC implementation in the Appendix A.

4 Performance

Our primary goal is to perform a comparative analysis between plain-DNS, SK-DNSSEC and PK-DNSSEC, in order to evaluate the performance tradeoff induced by the security overhead.

In this paper, we consider the public-key DNS security extensions (PK-DNSSEC) defined in RFC2535 [7]. There are several work-in-progress IETF drafts [17, 18] that will eventually supersede RFC2535, however the results presented in this paper will still be valid since we are mainly concerned with performance evaluation. The most important change in these drafts is the addition of a *Delegation Signer* (`DS`) record that delegates trust from a parental key to a child's zone key. This simplifies key management, but it does not reduce the computational cost for a resolver and will not affect the overall performance.

4.1 Setup

For our experimental analysis, we have setup the DNS tree depicted in Fig. 1. Each of the domains corresponding to the nodes in the tree is hosted on a separate machine. The machines are part of a single Ethernet LAN segment. They reside on the same subnet, connected by a Trendnet TEG-S240TX Gigabit switch, with no intermediate routers in between. All machines have the same hardware configuration, namely single Athlon XP 2.2 GHz processor, 1 GB SDRAM memory, running Red Hat Linux 8.0 (kernel 2.4.20) and BIND 9.2.1 compiled with OpenSSL 0.9.7d.

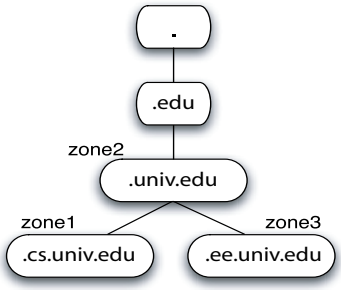

Figure 1: The test DNS tree

Each node in the DNS test tree is responsible for exactly one zone. The zones for `.` and `.edu` contain only one host as well as basic delegating records. The real zones used for testing are `.cs.univ.edu`, `.univ.edu` and `.ee.univ.edu` (zones 1, 2 and 3 respectively). Contents of these zones are elaborated on later, as they are adapted to the different types of tests performed.

In the case of PK-DNSSEC, the zones were signed using 1024-bit RSA keys (with public exponent $e = 2^{16}+1$, the default in OpenSSL), while for SK-DNSSEC we used AES 128-bit symmetric keys. All the machines used in the DNS tree have EDNS0 enabled [19], so the size of the DNS packets sent over UDP is not limited to 512 bytes. The EDNS0 capability is advertised in the DNS query, announcing that the requester can handle UDP payloads up to 4096 bytes. This enhancement does not help SK-DNSSEC, because the packets are usually smaller than 512 bytes, even with the addition of symmetric certificates. However, it greatly improves the performance of PK-DNSSEC, since otherwise requests greater than 512 bytes would mandate the use of TCP which is more resource-intensive than UDP. Even though the advertised payload for UDP is 4096 bytes, the IP layer still fragments the packets before sending them to the MAC layer, in order to fit the usual size of 1500 bytes for Ethernet packets. In the network traffic performance tests for PK-DNSSEC, some of the packets were bigger than 1500 bytes (thus causing fragmentation). However, the additional header bytes caused by the fragmentation were not counted towards the final results.

To compensate for the small number of name servers

in the test DNS tree, the TTL (time to live) values for zones were chosen smaller than values of realistic TTLs. This implies that the records expire faster. A small TTL forces name servers to query for records more frequently, thus effectively simulating a high workload which is closer to a real-world scenario. Experiments were performed with TTL values between 1 and 60 seconds. We argue that using small TTLs does not bias the results in favor of any of the considered models. We also argue that the results give an approximation of the behavior when larger TTLs are used. The experiments were conducted on a smaller scale and our goal was to give a preliminary performance evaluation. A more complete set of tests would involve the use of larger TTLs and a much larger number of DNS servers, needed to generate a high query rate.

4.2 Experiments

We group the performance tests in three categories: query throughput, network traffic and query latency. We believe that for network traffic performance evaluation, it is more important to simulate a realistic DNS traffic pattern than it is for the query throughput performance tests. Querying for different types of `RR`s has less impact on the query processing rate of a name server than on the size of DNS responses. Thus, for the network traffic tests, we chose to model the traffic pattern and zone contents after a real DNS trace. However, we did not follow the same principles for the query throughput evaluation because we wanted to minimize the influence of network overhead caused by larger DNS messages.

4.3 Query throughput

The query throughput of a DNS server is defined as how many queries per second the DNS server is capable of handling. Each of the zones 1, 2 and 3 contains 10,000 hosts and consists of one `SOA` resource record (`RR`), one `NS RR` and 10,000 `A RR`s (with distinct IP addresses chosen from different class B address pools). Since we are only going to query for `A RR`s, the zones do not have other types of resource records[2]. The workload for the name servers was generated using *queryperf*, a DNS query performance testing tool bundled with the BIND9 distribution [20]. A query throughput performance test is a stress test that measures the raw query throughput of a DNS server. *Queryperf* can have a certain amount of outstanding queries, and for the query throughput performance tests, we used the default setting of 20. To avoid additional load on the machines hosting the zones, *queryperf* was executed on a separate machine, outside the DNS test tree, but still inside the same Ethernet segment. In accordance with the SK-DNSSEC protocol, we also modified *queryperf* to include symmetric certificates. For PK-DNSSEC, *queryperf* was executed with the flag -D to ensure that DNSSEC records are requested. Furthermore, we verified that in all answers the authentic data bit (AD) was set, indicating that all authentications had been successful.

To maintain consistency in the comparative tests, each category of tests was run with the same batch of queries for all the three configurations of the DNS tree: plain-DNS, SK-DNSSEC and PK-DNSSEC. The tests in different categories were executed independently from each other, by restarting all the name servers between executions. For each category of tests, the results were averaged over a set of ten measurements. The experimental results are described in the next sections and analyzed in Section 4.3.4.

4.3.1 Performance of a recursive (caching) DNS server.

When a caching name server answers a query from the cache, it requires much less CPU time and fewer packets of network traffic than when it answers a query for which the server needs to perform a recursive lookup by querying authoritative servers. Therefore, just like in [21], we characterize the throughput of a caching server by two numbers: (1) the throughput when the answers are not in the cache, and (2) the throughput when answers are already in the cache. The actual throughput with a mixed production load will be somewhere between these two numbers, closer to one or the other depending on the cache hit rate.

Our basic query batch consists of 10,000 queries, matching all the `A RR`s in *zone3* (`.ee.univ.edu`). Using *queryperf*, the queries were directed to the name server authoritative for *zone1* (`.cs.univ.edu`), which played the role of a caching resolver. This means that a query for `host1.ee.univ.edu` will require two queries to the authoritative servers: one to the `univ.edu` server returning a referral and one to the `ee.univ.edu` server returning the answer. We designed this test in order to simulate the behavior of typical web surfing clients: a typical lookup for the uncached web server address `www.domain.com` requires a query to the `com` server and one to the `domain.com` server.

Table 2 shows the results when the answers are entirely uncached and entirely cached. These results were obtained using the basic query batch and are in-

[2]This simplifies the measurement process and minimizes the influence of network overhead caused by *RR*s of larger size.

Configuration	uncached (qps)	cached (qps)
plain-DNS	2550	17800
SK-DNSSEC	1860	17793
PK-DNSSEC	1313	17779

Table 2: Caching server performance for entirely uncached and entirely cached answers

Configuration	authoritative (qps)	referral (qps)
plain-DNS	18070	17200
SK-DNSSEC	8633	5206
PK-DNSSEC	11440	13535

Table 3: Authoritative and referral server performance (in queries per second)

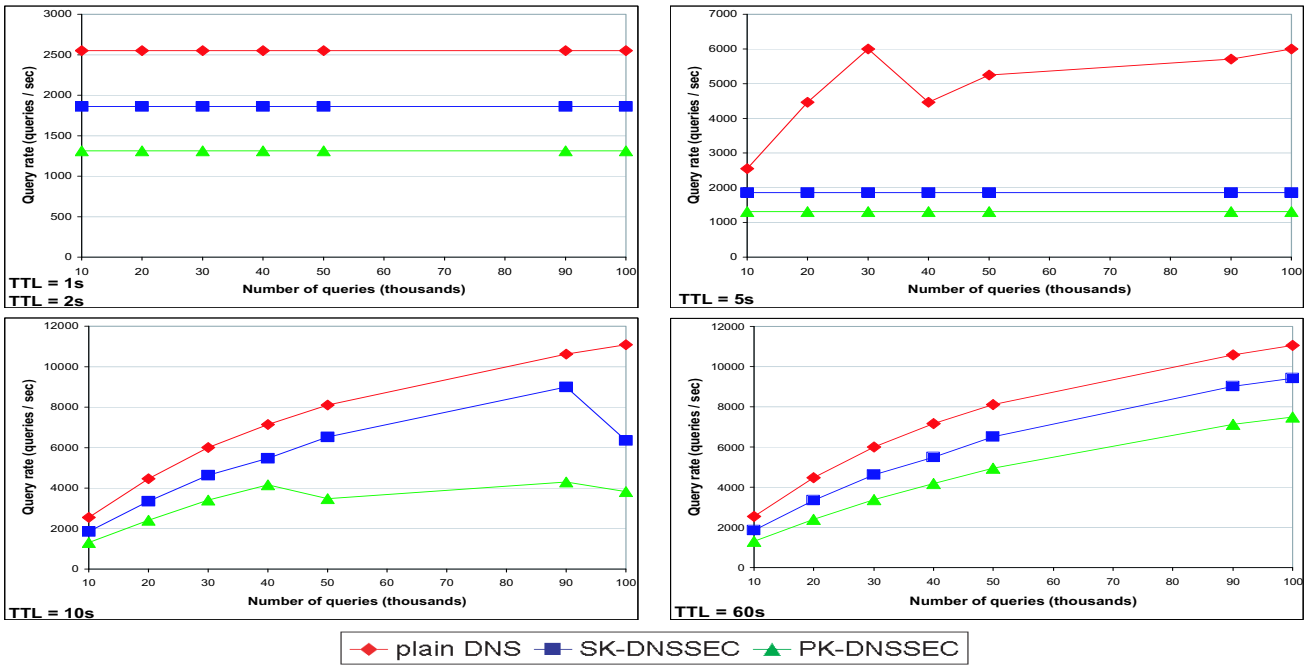

Figure 2: Query throughput performance for a caching server, for various zone TTLs, averaged over ten measurements (note that each graph has a different scale)

dependent of the zone TTL. In addition to the basic query batch, we performed tests with query batches of n thousand queries, with $n \in \{20, 30, 40, 50, 90, 100\}$, obtained by repeatedly concatenating the basic query batch of 10,000 queries. Using these batches we simulated a mixed production load, where some of the queries will be answered from the cache, depending on the zone TTL. Results are shown in Fig. 2.

4.3.2 Performance of an authoritative and referral DNS server.

To test the performance of an authoritative name server, the query batch was constructed by choosing hosts from *zone1* and then directed to the name server authoritative for *zone1*. In the case of a referral server, the queries were for hosts from *zone1* and were directed to the name server authoritative for *zone2*. This name server had recursion turned off and made referrals by answering with the data it had about the name server authoritative for *zone1* (the answer consists of `NS RRs`, called *delegation points*, and `A RRs`, called *glue addresses* [7]). For the SK-DNSSEC tests, a modified version of *queryperf* was used, in order to ensure that symmetric certificates are validated, new ones are created in case of referrals, and answers are authenticated for authoritative answers, as required by the SK-DNSSEC protocol. Table 3 shows the results. Note the results are independent from the zone TTL, as they are not influenced by caching.

4.3.3 Performance of a root DNS server.

In the case of SK-DNSSEC, the root name server receives requests for root certificates and this is the only step in which public-key cryptography is used. To determine the rate at which a name server can handle root certificate requests, we directed the query batch

to the root name server in the test DNS tree. Once more, the modified version of *queryperf* was used, to include root certificate requests. The root name server was able to handle approximately 305 root certificate requests per second[3]. We believe this is acceptable since root certificates are requested only once (when resolvers become operative for the first time) and their validity can be set arbitrarily long[4]. It is important to stress that root certificate requests are distinct from root DNS queries: A root server may receive millions of DNS queries but may have to handle only a small number of root certificate requests. An alternative is to deploy PK-DNSSEC at the top level of the DNS tree and SK-DNSSEC below. This would distribute the load among several servers. Another possibility is to deploy standard mechanisms to prevent denial-of-service attacks at the root, such as client puzzles [22, 23, 24].

4.3.4 Query Throughput Performance Analysis.

All the query throughput percentage values described in this section are expressed relative to the plain-DNS results. The most significant burden for PK-DNSSEC is on recursive name servers that act as caching resolvers. Observe in Table 2, for uncached queries, that SK-DNSSEC causes a smaller decrease of the query throughput performance of a caching DNS server than PK-DNSSEC does: while SK-DNSSEC stays at 73%, PK-DNSSEC gets as low as 51%. If the queried name server is not authoritative for a queried domain and if a query is not cached, then a portion of the DNS tree is traversed during the resolving process. Thus, testing for the performance of a recursive server is a good indicator for the performance of the whole DNS tree, since it also involves referrals and authoritative answers in addition to purely recursive answers. Since PK-DNSSEC performs better at answers that are purely referral or authoritative, the difference in performance for a recursive server can only be attributed to the additional burden placed on resolvers by PK-DNSSEC. Indeed, for PK-DNSSEC, caching resolvers have to verify the signed answers, which involves a public-key verification; for SK-DNSSEC, caching resolvers only have to send an already pre-computed symmetric certificate, thus no cryptographic operations are necessary.

In Fig. 2, for the zone TTL of 1 and 2 seconds, caching was not effective for any of the tested configurations. Indeed, it took plain-DNS 3.9 seconds to finish querying for all the 10,000 hosts in the query batch, while it took 5.3 seconds for SK-DNSSEC and 7.6 seconds for PK-DNSSEC. For zone TTL of 5 seconds, we start to see the effects of caching for plain-DNS, while for SK-DNSSEC and PK-DNSSEC, caching is visible only starting with zone TTL of 10 seconds. Observe that in some cases, when the number of queries increases, the query rate drops; take for example the case of TTL=5s, plain-DNS, from 30000 to 40000 queries. The explanation is that resolving 30000 queries falls just inside the 5 seconds interval, which is the zone TTL, and the additional 10,000 queries are treated as uncached queries, thus lowering the average query rate.

PK-DNSSEC is vulnerable to a particular type of denial-of-service attack when a caching resolver does not receive the typical mix of queries, but rather a stream of queries for non-existent hostnames. This causes the generation of a stream of signed non-existence records, which are more expensive to validate. After additional tests, we concluded that such an adversarial query batch decreases the performance of a caching resolver by 20% for PK-DNSSEC. Plain DNS and SK-DNSSEC are not vulnerable to this attack.

On the other hand, Table 3 shows that PK-DNSSEC performs better than SK-DNSSEC for an authoritative name server (63% compared to 47%) and for a referral name server (78% compared to 30%). The increased performance for authoritative answers was expected, since PK-DNSSEC needs no additional computations, while for SK-DNSSEC a symmetric certificate needs to be verified and a MAC needs to be computed. Similarly, for referral answers, PK-DNSSEC only serves the pre-signed data, while SK-DNSSEC needs to verify a symmetric certificate, create a new one, and also encrypt and authenticate a new pair of symmetric keys. Also, we suspect that the difference in performance between authoritative and referral answers for PK-DNSSEC is caused by the additional *RR*s present in authoritative answers.

It is worth mentioning that, according to our measurements, the cryptographic operations in the SK-DNSSEC implementation accounted for only a small percentage of the total cost added by SK-DNSSEC (28% for the referral name server test and 26% for the authoritative name server test). The overhead seems to be mostly caused by the rest of the code (data structures handling, DNS message re-parsing etc), that can potentially be optimized in future releases, thus further

[3]Incidentally, notice that this is approximately the same number of queries a PK-DNSSEC-enabled server would be able to handle if signatures would have to be computed on the fly and it is the main reason why we argue that PK-DNSSEC is not suitable for dynamic environments.

[4]More specifically, a root certificate can be valid, for example, for 6 months or 1 year or more given that the public keys of root servers are chosen to last for a long period of time. In principle, a root certificate is valid as long as the public key of the root server is not changed.

improving the performance of SK-DNSSEC.

4.4 Network traffic

In this section we are interested in measuring the network traffic generated under the plain-DNS, SK-DNSSEC-enabled and PK-DNSSEC-enabled models.

4.4.1 Testbed setup.

To obtain a realistic query type and query outcome distribution for our query batch, we have monitored the DNS network traffic at the main DNS server of our institution. The data was recorded for 8 consecutive days, 8 hours daily, between 8AM-4PM. In this interval more than four million queries were observed, with the query type and query outcome distributions as shown in Table 4 and Table 5, respectively. The query type distribution in Table 4 is consistent with the numbers observed in [25] for a root server, and [26] for the MIT LCS and AI labs[5].

Query Type	**Percentage** (%)	**Query Type**	**Percentage** (%)
A	60.452	SOA	0.111
PTR	16.605	SRV	0.093
AAAA	15.164	NS	0.042
MX	7.311	other	<0.010
A6	0.211		

Table 4: Observed query type distribution

The query type distribution is relevant when evaluating the network traffic because queries of different types can result in differently sized answers. Also, the query outcome distribution plays an important role if we consider, for example, the cost of processing queries for non-existent hostnames in the case of PK-DNSSEC: validation for signed negative answers is usually more expensive than for signed positive answers. Thus, we considered both query type distribution and query outcome.

While trying to maintain the same query type distribution as in Table 4 for the query batch, a few changes were made that should have a negligible impact on the performance results: Instead of queries for `PTR` records, we used queries for `A` records. This should not make a difference since an answer to a query for a reverse address mapping (`PTR`) record has about the same size of an answer to a query for a regular address (`A`) record. Also, since all the other types of resource records, besides `A`, `PTR`, `AAAA` and `MX` account for less than 0.5% of the total number of queries, we argue they have a negligible impact and we do not include them in our query batch.

In addition, resemblance to a real-world scenario was considered for the contents of the zones in the DNS tree. The test zones consist of one `SOA` resource record (`RR`), two `NS RR`s and one `A RR` for each of the 10,000 hosts in the zone. It is a common practice to have at least two `NS RR`s per zone and two `MX RR` per domain for redundancy reasons. With only three test zones in our DNS tree, having only two `MX RR`s per zone causes an overwhelming majority of queries for `MX RR`s to be answered from the cache. To avoid this and simulate what happens in the real DNS with a much larger name space, we assigned two `MX RR`s to 1000 of the hosts in each zone. This setting is satisfactory given the amount of MX records (over 7%) in the query batches.

Query Type	**Percentage(%)**					
	success	referral	nxrrset	nxdomain	failure	total
A	55.26	<0.01	<0.01	4.97	0.21	60.45
PTR	15.35	<0.01	<0.01	1.02	0.23	16.60
MX	5.97	<0.01	1.11	0.08	0.15	7.31
AAAA	0.61	<0.01	11.48	1.55	1.52	15.16
Total	77.19	0.01	12.59	7.62	2.11	99.52

Table 5: Observed query outcome distribution: *'success'* represents successful queries the name server handled that did not result in referrals or errors; *'referral'* are the queries that resulted in referrals; *'nxrrset'* are the queries that resulted in error responses because the queried domain existed, but the queried resource record did not exist for that domain; *'nxdomain'* are queries that resulted in error responses because the queried domain did not exist; *'failure'* are the queries that resulted in errors other than those covered by *'nxrrset'* and *'nxdomain'*.

4.4.2 Network Traffic Performance Tests.

Using the same DNS test tree, a batch of 10,000 queries was directed to the name server responsible for *zone1*. The queried domains were chosen from *zone3*, while the query type distribution followed the description in Section 4.4.1 and Table 4. The percentage of queries that resulted in error (*nxrrset+nxdomain+failure*) is considerable (over 22%), and we paid special attention to include queries with such outcome in the query

[5]The only exception is the large number of AAAA queries in our trace, which we suspect occurred because of a bug in version 8.12.9 of `sendmail`: IPv6 DNS lookups are attempted before IPv4 lookups, even if IPv6 is not enabled in the kernel of the operating system.

Configuration	TTL (sec)	Network traffic average (KB/sec)	Queries resolved	Configuration	TTL (sec)	Network traffic average (KB/sec)	Queries resolved
plain-DNS	1s	652	20283	plain-DNS	25s	639	243964
SK-DNSSEC	1s	733	16283	SK-DNSSEC	25s	669	233482
PK-DNSSEC	1s	1724	11761	PK-DNSSEC	25s	1844	231766
plain-DNS	10s	726	79845	plain-DNS	60s	384	738013
SK-DNSSEC	10s	768	69198	SK-DNSSEC	60s	391	699924
PK-DNSSEC	10s	1730	34330	PK-DNSSEC	60s	1024	694919

Table 6: Network traffic statistics

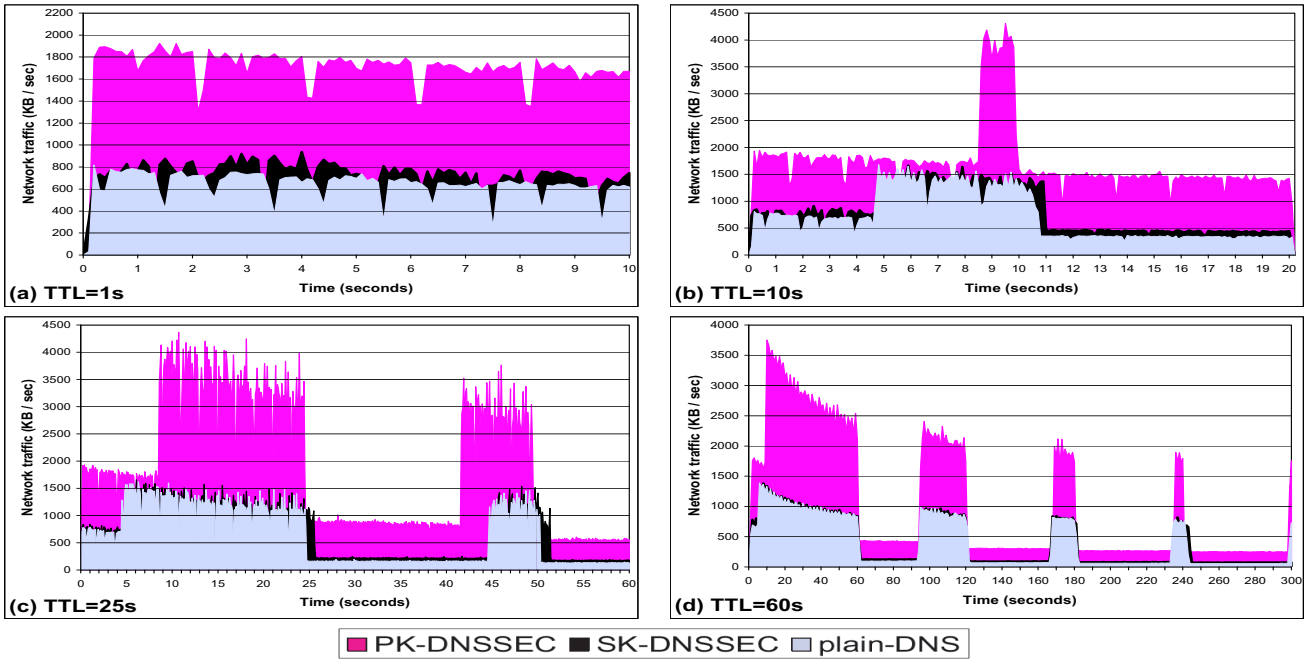

Figure 3: Network traffic evolution over time (note that each graph has a different scale)

batch[6], according to the data in Table 5. The queries were run from a machine outside the test DNS tree, but still inside the same Ethernet segment.

Queryperf was used to generate the workload, with the default setting of 20 outstanding queries, for intervals of i seconds, with $i \in \{10, 20, 60, 300\}$. As a result, during an interval, it was possible for the query batch to be executed multiple times for some configurations of the name servers in the DNS tree. The results were gathered using *tcpdump* from yet another machine outside the DNS tree, but inside the same Ethernet segment. Tests were run with the zones in the DNS tree having a TTL of t seconds, with $t \in \{1, 10, 25, 60\}$. Results are aggregated in Fig. 3 and summarized in Table 6.

With $t = 1$ and $i = 10$ (Fig. 3(a)), SK-DNSSEC averages to 733 KB/sec, relatively close to the average of 652 KB/sec for plain-DNS. In contrast, PK-DNSSEC imposes a much higher bandwidth with an average of 1724 KB/s. Moreover, during the test interval, the SK-DNSSEC resolver was able to complete 80% of the number of queries resolved by plain-DNS, as opposed to only 57% in the case of PK-DNSSEC. Thus, for SK-DNSSEC, not only was the amount of traffic generated much smaller than for PK-DNSSEC, but also the number of resolved queries was considerably larger.

With $t = 10$ and $i = 20$ (Fig. 3(b)), we observe an

[6]To generate a *failure* outcome for a query, we create a lame delegation of a domain, and ask a query for a host in that domain. For outcomes such as *nxrrset* and *nxdomain*, we query for a non-existent `RR` set or hostname.

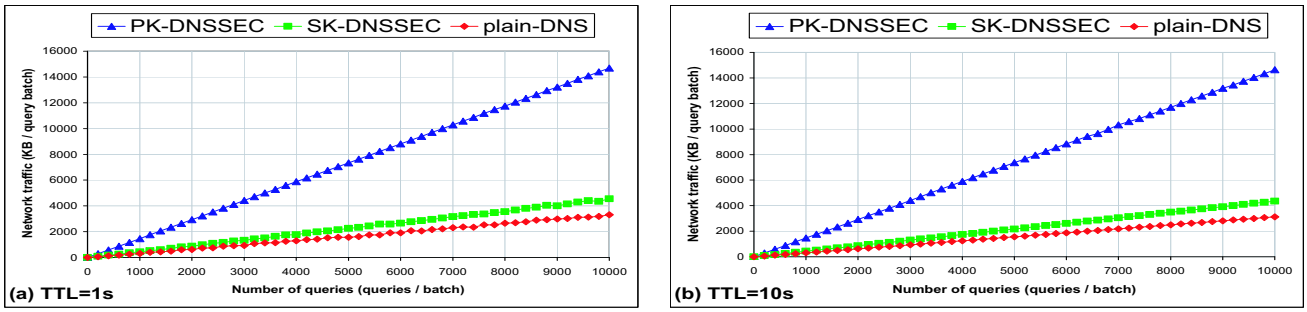

Figure 4: Network traffic evolution with query volume

interesting behavior. After the batch of 10,000 queries is exhausted between seconds 5 and 6 for plain-DNS and SK-DNSSEC, *queryperf* runs the query batch multiple times during the specified interval, and since the TTL of the zones is now 10s, after this moment all the queries in the batch are already cached. That explains the sudden increase in network traffic and in number of queries resolved: The resolver is able to answer from the cache a higher number of queries, thus generating more traffic (mostly between the resolver and the client that issued the queries). After the 10 second mark, we observe a gradual decrease in the amount of network traffic, as the TTL of the cached records expires. The same caching behavior is seen for PK-DNSSEC, but for a shorter interval, since it took more time to resolve the first 10,000 queries.

With $t = 25$, $i = 60$ (Fig. 3(c)) and $t = 60s$, $i = 300$ (Fig. 3(d)), the caching behavior becomes more obvious; also, as the TTL increases, the number of resolved queries converges to the same value for the three models.

In all cases, and especially for larger TTLs, we observe that PK-DNSSEC generates a considerably larger amount of network traffic than plain-DNS (between 164%-188% more traffic), while SK-DNSSEC stays relatively close to plain-DNS (between 1%-12% more traffic). This considerable difference is caused by the large message size in the PK-DNSSEC model.

While Fig. 3 is illustrative for the evolution of network traffic over time, it does not fully capture the relation between the amount of traffic generated and the number of queries. To better illustrate this relationship, additional experiments were performed, using the same setup for the zones in the DNS tree as in the previous network traffic experiment; however, this time the tests were run with batches of n queries, with $n \in \{100, 200, 300, ..., 9900, 10000\}$. All these query batches had the distributions described in Table 4 for query type and Table 5 for query outcome. We used *queryperf*, but with only one outstanding query, because this setting generates more accurate results for network traffic measurements. Tests were run with the zones in the DNS tree having a TTL of 1 second and 10 seconds. The results were collected with *tcpdump* and aggregated in Fig. 4.

As we increase the size of the query batches, we observe that the amount of traffic induced by SK-DNSSEC remains in close range to plain-DNS, while for PK-DNSSEC it increases at a much faster rate. When increasing the zone TTL, we observe a slight decrease in the amount of network traffic, which can be explained because delegation point `NS` and glue `A RR`s are cached longer.

Note that PK-DNSSEC notably increases the size of DNS response packets; among other issues, this confirms that PK-DNSSEC-aware servers can act as denial-of-service amplifiers, as hypothesized in [4].

4.5 Query Latency

The query latency of a caching DNS server is the time it takes to answer any single DNS query. It can be a real issue for DNS, since it is the aspect of server performance that is most visible to the individual end user. Another experiment was run to evaluate the query latency. The name servers used for this test were configured as shown in Fig. 1, but were physically located so that realistic network delays were involved: the name server authoritative for *zone1* was part of one network, while the rest of the name servers were part of a different network; the two networks were 17 hops away, the first one located in Italy, and the second one located in the USA. In the test, queries for hosts in *zone3* were directed to the name server authoritative for *zone1*, which played the role of a resolver. The answers for these queries were not previously cached and the results in Table 7 are averaged over a set of 100 queries.

The latency for SK-DNSSEC is slightly higher than for plain-DNS, since additional cryptographic operations are involved in the process of query resolving. On

the other hand, SK-DNSSEC has a lower query latency than PK-DNSSEC, mainly because in PK-DNSSEC the resolver has to contact the name servers in the DNS tree twice: once to get the actual signed answer and once to get the key material required to validate the answer. SK-DNSSEC, just like plain-DNS, only contacts the intermediate servers once. Basically, if the answer is not already cached, then the round trip time between name servers involved in the resolving process has a higher influence over the query latency in PK-DNSSEC than in SK-DNSSEC. Note that the query latency will not significantly depend on the speed of the machines running the name servers, because it is dominated by external network delays rather than by processing time [21].

The case where the answer is not previously cached represents an upper bound on the query latency. Obviously, depending on the amount of caching available at the resolver or at intermediate name servers, the query latency gap between PK-DNSSEC and SK-DNSSEC can be smaller.

Configuration	Latency (milliseconds)
plain-DNS	505.76
SK-DNSSEC	509.70
PK-DNSSEC	1360.82

Table 7: Query latency for non-cached answers (averaged over 100 queries)

5 Remarks

We saw that the computational cost of adding security to DNS is different depending on the type of name server. If a hybrid approach is considered, with PK-DNSSEC deployed for the top-level domains, where the information is static, then SK-DNSSEC would be suitable for the low-level DNS tree, which is characterized by a more dynamic environment. Such a hybrid approach has several positive aspects. Our experiments showed that the computational cost of PK-DNSSEC is high for caching resolvers, while SK-DNSSEC places most of the computational cost on non-recursive servers above the zone that is being searched. Thus, PK-DNSSEC pushes the computational cost towards the bottom of the DNS tree, while SK-DNSSEC pushes it upwards. A hybrid approach would eliminate these shortcomings: with PK-DNSSEC on top, referrals are efficient (which is important for servers that handle high-volume traffic), while SK-DNSSEC on the bottom reduces the computational burden for caching resolvers (since resolvers are usually at the bottom of the hierarchy).

We also noticed that the cryptographic core of the signing routine in SK-DNSSEC is responsible only for a fraction of the total cost incurred in generating symmetric certificates and that its performance can be further improved by employing faster cryptographic primitives. For instance, one could substitute HMAC with UMAC which appears to be one order of magnitude faster [27].

While experimenting with the three versions of DNS, we have analyzed some aspects and considered techniques we plan to include in a future release of the code. In particular, one issue we are addressing is the fact that pre-computation in SK-DNSSEC is not possible since authentication is always achieved via freshly generated secret keys. This offers a high level of security against replay attacks but it requires secret keys to be stored on-line so that they are readily available to the DNS server. This does not apply to PK-DNSSEC since signatures are pre-computed over entire *RR* sets and re-used until they expire. However, key management in PK-DNSSEC is a big issue, particularly in the case of *dynamic updates* [28], and it appears that the only way to effectively address it is to have certain keys online. We are planning to devise techniques to mitigate this online-key issue with a combination of intrusion detection and proactive security mechanisms [29].

Finally, we are addressing the fact that SK-DNSSEC employs public-key cryptography whenever a root symmetric certificate is needed either because a new resolver is being set up or because an existing root certificate has expired. In both cases, we argued that a root server can handle the load caused by legitimate requests but an SK-DNSSEC-enabled root server is potentially susceptible to a denial of service attack. In a future release of the code we are planning to incorporate the following strategy which may mitigate the issue above: The root private key is kept off-line and root certificate requests are only collected and later elaborated offline at certain time intervals. The delay between the request and the response from the root server could be fixed and predetermined. In this way, a resolver with an expiring certificate will have a time window before the expiration date in which it is allowed to request the new certificate. This should be enough to limit service disruptions. Alternatively, we are also looking at mechanisms based on client puzzles [22, 23, 24] but tailored to the specific needs of DNS.

6 Conclusions and Future Work

In this paper, we have presented a functional implementation of the SK-DNSSEC protocol and we have performed a comparative analysis between plain-DNS, SK-DNSSEC and PK-DNSSEC in order to evaluate the performance tradeoff induced by the security overhead.

We saw that a hybrid approach, with PK-DNSSEC deployed for top-level domains and SK-DNSSEC for the low-level DNS tree, can leverage the benefits of both security extensions. PK-DNSSEC significantly increases the size of DNS response packets, generating considerably more network traffic and higher network latency than plain-DNS or SK-DNSSEC. In general, SK-DNSSEC appears to be a valid alternative to PK-DNSSEC since it improves on several other important aspects. For instance, it simplifies key management, it is less intrusive than PK-DNSSEC, given that zone files do not have to be changed, and no `NXT RR`s are needed. In addition, since response packets in SK-DNSSEC are smaller, less memory for caching is required.

Availability. The implementation of the SK-DNSSEC-enabled BIND9 name server is available at `http://skdnssec.isi.jhu.edu`. The code we released implements the basic SK-DNSSEC scheme, and should be considered a preliminary version. We encourage and appreciate any feedback.

Acknowledgments. We are grateful to Fabian Monrose for his insightful comments that notably improved this paper. We thank Daniel Massey, Adam Stubblefield, Breno de Medeiros, Kevin Fu and Emil Sit for their feedbacks on the paper. Many thanks to Steve Rifkin for his assistance in collecting DNS traffic data. We are also grateful to Scott Rose for his suggestions on setting up the performance testbed. We thank the anonymous reviewers for their helpful comments. This work was supported by an NSF grant.

References

[1] R. Curtmola, A. Del Sorbo, and G. Ateniese, "On the performance and analysis of DNS security extensions," in *Proceedings of the Fourth International Conference on Cryptology and Network Security (CANS '05)*, Lecture Notes in Computer Science, Springer-Verlag, 2005.

[2] G. Ateniese and S. Mangard, "A new approach to DNS security (DNSSEC)," in *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pp. 86–95, ACM Press, 2001.

[3] S. M. Bellovin, "Using the Domain Name System for system break-ins," in *Proceedings of the Fifth USENIX UNIX Security Symposium*, pp. 199–208, June 1995.

[4] D. Atkins and R. Austein, *Threat Analysis Of The Domain Name System.* IETF - Network Working Group, August 2004. RFC3833.

[5] P. Vixie, "DNS and BIND security issues," in *Proceedings of the Fifth USENIX UNIX Security Symposium*, pp. 209–216, June 1995.

[6] T. de Raadt, N. Provos, T. Miller, and A. Briggs, "Bind vulnerabilities and solutions," April 1997. `http://niels.xtdnet.nl/papers/secnet-bind.txt`.

[7] D. Eastlake, *Domain Name System Security Extensions.* IETF - Network Working Group, March 1999. RFC2535.

[8] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology - Crypto '96 Proceedings* (N. Koblitz, ed.), vol. 1109 of *LNCS*, Springer-Verlag, 1996.

[9] H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication.* IETF - Network Working Group, February 1997. RFC2104.

[10] D. Davis and R. Swick, "Network security via private-key certificates," in *Proceedings of the Third USENIX UNIX Security Symposium*, pp. 239–242, September 1992. Also in ACM Operating Systems Review, v. 24, n. 4 (Oct. 1990).

[11] B. C. Neuman and T. Ts'o, "Kerberos: An authentication system for computer networks," in *IEEE Communications*, vol. 32(9), pp. 33–38, IEEE, September 1994.

[12] J. Kohl and C. Neuman, *The Kerberos Network Authentication System (V5).* IETF - Network Working Group, September 1993. RFC1510.

[13] V. Ramasubramanian and E. Sirer, "The design and implementation of a next generation name service for the internet," in *Proceedings of SIGCOMM'04*, ACM, 2004.

[14] X. Wang and H. Yu, "How to break MD5 and other hash functions," in *Proceedings of EuroCrypt 2005*, vol. 3494 of *Lecture Notes in Computer Science*, pp. 19–35, Springer-Verlag, 2005.

[15] A. Lenstra and B. de Weger, "On the possibility of constructing meaningful hash collisions for public keys," in *Proceedings of ACISP 2005*, vol. 3574 of *Lecture Notes in Computer Science*, pp. 267–279, Springer-Verlag, 2005.

[16] D. Whiting, B. Schneier, and S. Bellovin, "AES key agility issues in high-speed IPsec implementations."

[17] R. Arends, M. Larson, R. Austein, D. Massey, and S. Rose, "Protocol modifications for the DNS security extensions," Internet draft 09, IETF - DNS Extensions, October 2004.

[18] R. Arends, M. Larson, R. Austein, D. Massey, and S. Rose, "Resource records for the DNS security extensions," Internet Draft 11, IETF - DNS Extensions, October 2004.

[19] P. Vixie, *Extension Mechanisms for DNS (EDNS0)*. IETF - Network Working Group, August 1999. RFC2671.

[20] "BIND." `http://www.isc.org/sw/bind`.

[21] NOMINUM, "How to Measure the Performance of a Caching DNS Server," 2002. `http://www.nominum.com/content/documents/CNS_WP.pdf`.

[22] D. Dean and A. Stubblefield, "Using client puzzles to protect TLS," in *Proceedings of the Tenth USENIX Security Symposium*, August 2001.

[23] A. Juels and J. Brainard, "Client puzzles: A cryptographic defense against connection depletion attacks," in *Proceedings of NDSS '99* (S. Kent, ed.), pp. 151–165, 1999.

[24] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten, "New client puzzle outsourcing techniques for DoS resistance," in *11th ACM Conference on Computer and Communications Security (CCS 2004) (to appear)*, ACM, 2004.

[25] D. Wessels and M. Fomenkov, "Wow, that's a lot of packets," in *Proceedings of Passive and Active Measurement Workshop (PAM2003)*, April 2003.

[26] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, "DNS performance and the effectiveness of caching," in *Proceedings of the ACM SIGCOMM Internet Measurement Workshop '01*, (San Francisco, California), November 2001.

[27] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway, "UMAC: Fast and secure message authentication," in *Advances in Cryptology - Crypto '99 Proceedings*, vol. 1666 of *LNCS*, pp. 216–233, Springer-Verlag, 1999.

[28] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound, *Dynamic Updates in the Domain Name System (DNS UPDATE)*. IETF - Network Working Group, April 1997. RFC2136.

[29] R. Ostrovsky and M. Yung, "How to withstand mobile virus attacks," in *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 51–59, ACM Press, 1991.

[30] "BIND 9.2.1 source code." `ftp://ftp.isc.org/isc/bind9/9.2.1/bind-9.2.1.tar.gz`.

[31] P. Vixie, O.Gudmundson, D. Eastlake, and B. Wellington, *Secret Key Transaction Authentication for DNS (TSIG)*. IETF - Network Working Group, May 2000. RFC2845.

A SK-DNSSEC Implementation

In order to understand the functionality provided by BIND, we first present an overview of how BIND handles DNS messages and provide details about its implementation. We then concentrate on the specifics of SK-DNSSEC, including an overview of how SK-DNSSEC handles messages, followed by implementation details.

A.1 BIND overview

Every name server running BIND can be viewed as made of two major entities: a server component (handling incoming requests and outgoing responses) and a resolver component (handling outgoing requests and incoming responses). The two entities usually interact as parts of different name servers, but can also interact on the same DNS server when the request received by the server component needs recursion in order to be resolved. This is usually the case when a DNS server acts as a resolver on behalf of a stub resolver. In this case, the server component calls the resolver component and waits until it resolves the request. Once the answer is received, the server component can respond to the stub resolver (or other DNS server) that initially queried it.

The BIND name server is a complex program with its own tasks, threads, scheduler, event dispatcher and memory management. It has managers handling most of its units, including a task manager, an interface manager, an event dispatch manager and a client manager. In what follows we focus on the client manager, which is responsible for most of the relevant message handling. A client manager is bound to each network interface and handles messages received on that interface. The description below is based on a direct examination of BIND's source code [30].

We now describe what happens inside the server and resolver components during the lifetime of a query. Assume that a stub resolver sends a query to its local DNS server. Whenever the local name server receives a request, the interface manager dispatches an event and the control passes to the server component. The client manager, as part of the server component, receives the event and a *client object* is created to handle it, if no existing *client object* can be reused. The *client object* has now a life of its own and will take all the necessary steps in order to resolve the request.

The *client object* parses the request and verifies the public key DNSSEC signature, if any. If the DNSSEC verification is successful, the *client object* looks for a response in the cache and in the zone for which the server is authoritative. If no answer is found, a recursion process is started, and the control passes to the resolver component. A *fetch object* is created to control the recursion process, as part of the resolver component. The *fetch object* traverses the DNS tree from the root down to the leaves, issuing iterative queries to the appropriate name servers. When the final answer is received, the control is returned to the server component, which sends it back to the stub resolver.

R_i	DNS server i
K_{XY}	secret key pair (encryption key, MAC key) shared by X and Y
P_{XY}	symmetric certificate shared by X and Y
$Nonce_i$	nonce sent to R_i
certificate list	list of symmetric certificates
server key list	list of keys shared by a resolver with the external name servers
client key list	list of keys shared by a name server with the resolvers
shared key list	list of keys shared by a name server with its parent and its children
temporary key list	list of keys sent along with the root certificate request
nonce list	list of nonces

Table 8: Notation

A.2 The implementation of SK-DNSSEC in BIND

PK-DNSSEC authenticates DNS messages by including new types of resource records in the message, such as public-key signature *RR*s. Instead, we have chosen to let SK-DNSSEC handle DNS messages as opaque data and authenticate them by appending binary data at the end of the message[7]. Non SK-DNSSEC-aware DNS servers can simply ignore (or log the existence of) this additional data, thus backward compatibility is provided.

The implementation of the SK-DNSSEC protocol has been adapted to the structure of a DNS server. As shown in Fig. 5, each of the server and resolver components can be further split into a receiving and a sending part. Thus, the completion of a query-answer cycle can be seen as a loop of four steps. To enable the SK-DNSSEC extensions, BIND9 was modified in these four parts:

1. **Outgoing query processing**, when the resolver sends the query; handled by the function `sk_query_put`

2. **Incoming query processing**, when the server receives the query; handled by the function `sk_query_verify`

3. **Outgoing answer processing**, when the server replies with the answer; handled by the function `sk_answer_sign`

4. **Incoming answer processing**, when the resolver receives the answer; handled by the function `sk_answer_verify`

[7]In a future release, we plan to implement a more elegant solution and store the additional authenticating SK-DNSSEC binary data as a "meta-record", à la `TSIG` [31].

On each name server, several additional data structures are required, including a list of symmetric certificates, a list of server keys, a list of nonces, a list of client keys, a list of shared keys and a list of temporary keys. For ease of exposition, let's assume that a stub resolver sends a DNS query to the SK-DNSSEC-aware local name server for the non-cached `A` resource record of `www.foo.com` (refer to Fig. 5). We adopt the notation shown in Table 8.

Outgoing query processing. To resolve the query, the local DNS server U starts the recursion process. After the resolver component of U has constructed the query message DNS_Req, and before sending it out to an external name server R_i, a symmetric certificate P_{R_iU} needs to be appended to the message. This enables U to securely communicate with R_i. The resolver component of U retrieves P_{R_iU} from the *certificate list*, generates a new nonce $Nonce_i$ and appends both of them at the end of the DNS_Req message. If there is no symmetric certificate for R_i, or if P_{R_iU} has expired, then a root certificate request RC_Req is generated and appended at the end of DNS_Req. In this case, a new key pair K_{temp} is also generated and the query will be redirected to the root name server R_0. K_{temp} is temporary and will be replaced later with the key pair K_{R_0U} sent by R_0. For now, K_{temp} is stored in the *temporary key list* along with the IP address of R_0. $Nonce_i$ is stored in the *nonce list* along with the query ID and the IP address of R_i (or the IP address of R_0 in case of a root certificate request). At this point the resolver component sends out the query and waits for an answer.

Incoming query processing. After the remote server R_i (or R_0 for a root certificate request) has received and parsed DNS_Req, it proceeds with the SK-DNSSEC steps. In case the message contains the sym-

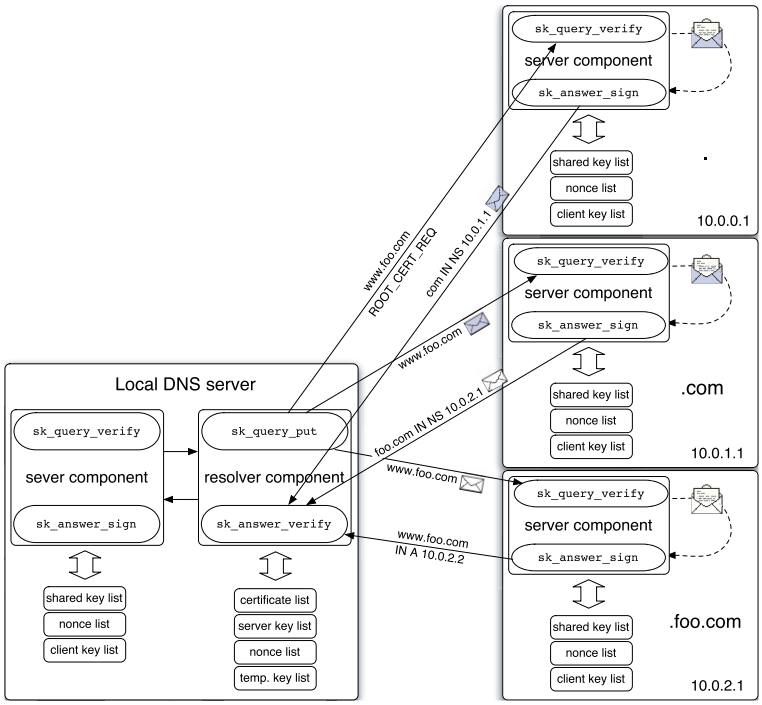

Figure 5: Example of an SK-DNSSEC-enabled query

metric certificate P_{R_iU}, it is decrypted and validated using the key pair $K_{R_{i-1}R_i}$, retrieved from the *shared key list*. In case the message contains the root certificate request RC_Req, it is decrypted using the root's private key. The key pair K_{R_iU} contained in P_{R_iU} (or the temporary key pair K_{temp} contained in RC_Req) is stored in the *client key list* along with the IP address of U, while the nonce $Nonce_i$ is stored in the *nonce list* along with the same IP address and the query ID. The remote server R_i then continues with the regular DNS operations until it acquires an answer DNS_Ans and needs to send it back to U.

Outgoing answer processing. To determine if DNS_Ans is a final answer or a delegating answer, the header of DNS_Ans is parsed. In both cases, $Nonce_i$ is retrieved from the *nonce list*, and the key pair K_{R_iU} is retrieved from the *client key list*. For a final answer, DNS_Ans is signed with a MAC function under the client key K_{R_iU}. For a delegating answer, the body of DNS_Ans is parsed to find out the IP address of R_{i+1} and a new key pair $K_{R_{i+1}U}$ is also generated and encrypted together with the MAC signature under K_{R_iU}. In addition to this encryption, a new symmetric certificate $P_{R_{i+1}U}$ is also created under $K_{R_iR_{i+1}}$ and appended to DNS_Ans ($K_{R_iR_{i+1}}$ is retrieved from the *shared key list*). $K_{R_{i+1}U}$ and $P_{R_{i+1}U}$ will be used by U to communicate securely with the delegated name server R_{i+1}. If DNS_Req carried RC_Req then a root certificate P_{R_0U} is constructed and added. At this point, both $Nonce_i$ and K_{R_iU} are deleted from their respective lists and the response message is sent back.

Incoming answer processing. When the local DNS server U receives the response message, the resolver component resumes the recursion process. After the regular BIND parsing of DNS_Ans, $Nonce_i$ is retrieved from the *nonce list*, and K_{R_iU} (or K_{temp} in case of a reply to a root certificate request) is retrieved from the *server key list* (or from the *temporary key list*, respectively). These two are needed to verify the MAC signature that ensures the integrity of DNS_Ans. If the response message contains a delegating answer, the body of DNS_Ans is parsed again to retrieve the IP address of R_{i+1}. Then $P_{R_{i+1}U}$ and $K_{R_{i+1}U}$ are extracted from the message and stored in the *certificate list* and the *server key list*, respectively. For the particular case in which the response message contains the root certificate P_{R_0U}, the resolver component authenticates P_{R_0U} and K_{R_0U} and stores them in the *certificate list* and *server key list*, respectively. As mentioned above, K_{R_0U} replaces the role of K_{temp}, which is deleted from the *temporary key list*. The resolver component then continues with its normal operation.