Deploying Cryptography in Internet-Scale Systems: A Case Study on DNSSEC

Hao Yang[1], Eric Osterweil[2], Dan Massey[3], Songwu Lu[2], Lixia Zhang[2]
IBM T. J. Watson Research Center, haoyang@us.ibm.com[1]
Computer Science Department, UCLA, {eoster, slu, lixia}@cs.ucla.edu[2]
Computer Science Department, Colorado State University, massey@cs.colostate.edu[3]

***Abstract*—The DNS Security Extensions (DNSSEC) are among the first attempts to deploy cryptographic protections in an Internet-scale operational system. DNSSEC applies well-established public key cryptography to ensure data integrity and origin authenticity in the DNS system. While the cryptographic design of DNSSEC is sound and seemingly simple, its development has taken the IETF over a decade and several protocol revisions, and even today its deployment is still in the early stage of rolling out. In this paper, we provide the first systematic examination of the design, deployment, and operational challenges encountered by DNSSEC over the years. Our study reveals a fundamental gap between cryptographic designs and operational Internet systems. To be deployed in the global Internet, a cryptographic protocol must possess several critical properties including scalability, flexibility, incremental deployability, and ability to function in face of imperfect operations. We believe that the insights gained from this study can offer valuable inputs to future cryptographic designs for other Internet-scale systems.**

I. Introduction

Cryptographic mechanisms can provide effective means to secure the Internet, and the DNS Security Extensions (DNSSEC) [2], [4], [3] are among the first attempts to add cryptographic protection into one of the Internet's core systems (the DNS). The goal of DNSSEC is to add data integrity and origin authenticity to DNS query replies so that users can verify that the answers received are indeed originated from the intended DNS server and have not been altered. Securing the DNS service not only can defeat emerging threats like DNS hijacking and cache poisoning but also may provide a foundation for deploying other cryptographic-based security applications. Specifically, the DNS can be used to store and serve the public keys of various entities. After twelve years of development efforts by IETF, the DNSSEC standards were finalized in March 2005 and a number of testbeds, pilot deployments, and services have been rolled out in the last few years [12], [11], [21], [10], [27].

In this paper, we take as inputs the discoveries and lessons accumulated by the DNSSEC development community and present the first systematic examination of the design, deployment and operational challenges DNSSEC has encountered over the years. However the goal of our study is not only to document how these challenges manifest, but more importantly to understand where they come from. We show that many challenges arise from a few fundamental factors: the large scale and distributed nature of the operational DNS system, the existence of DNS data caching, and the inherent heterogeneity in the operations of DNS by different autonomous administrations. Along the way, we also offer suggestions on how to address some of these open challenges using simple yet effective techniques.

The main contributions of this paper are three-fold. First, we document and classify the challenges in the DNSSEC deployment and operations; to date many of such issues have only been discussed in various informal channels (e.g., mailing lists, expired Internet drafts, or personal communications). Second, we critically analyze the continuous efforts in the DNSSEC community for addressing these operational challenges, and offer our own solutions for some of the open issues. Finally, and perhaps most importantly, we summarize the evolution of DNSSEC into a set of design lessons, which we hope can help the designs of other cryptographic systems to be deployed on the Internet.

The rest of the paper is organized as follows. Section II provides background information on the design of DNS and DNSSEC. Section III discusses the design and deployment challenges from DNSSEC's hierarchical PKI. Section IV analyzes the issues due to DNS caching. Section V identifies challenges from heterogeneous operational practices. Section VI reviews these operational issues in the context of actual DNSSEC deployment data, and Section VII discusses the root cause of these challenges and the importance of distributed monitoring. Section VIII discusses the related work. Finally, Section IX concludes the paper with several design lessons learned from our study.

II. Background

The Domain Name System (DNS) [18], [19] is a distributed database that maps host names such as `www.ucla.edu` to IP addresses and provides a wide range of other mapping services ranging from email to geographic locations. Virtually every Internet application relies on looking up certain DNS data. In this section we introduce a basic set of DNS terminology which is used throughout the text, followed by an overview of the DNS Security Extensions.

A. Domain Name System

All DNS data is stored in core data structure called a *Resource Record* (RR), and each RR has an associated name, class, and type. For example, an IPv4 address for `www.ucla.edu` is stored in an RR with name

`www.ucla.edu`, class IN (Internet), and type A (IPv4 address). The set of *all* RRs associated with the same name, class, and type is called an *Resource Record Set* (RRset). Since DNS resolvers issue queries for name, class, and type tuples, they are inherently querying for RRsets (and not individual RRs). For example, when a browser queries for ⟨`www.ucla.edu, IN, A`⟩, the reply will be the RRset for `www.ucla.edu` with *all* of the IPv4 addresses for that name. Thus, the smallest unit that can be requested in a query is an RRset, and all DNS actions including cryptographic signatures discussed later, apply to RRsets rather than individual RRs.

The global DNS is a distributed database organized in a tree structure. At the top of the tree, the root zone delegates authority to *top level domains* such as `.com`, `.net`, `.org`, and `.edu`. The `.com` zone then delegates authority to create `ibm.com`, `.edu` delegates authority to create `ucla.edu`, and so forth. The information repository that makes up the domain database is divided into sections called *zones*. Each zone belongs to a single administrative authority and is served by multiple authoritative name servers to provide name resolution services for all names in the zone. By definition, a zone can contain one or more connected domains in the DNS name tree; in practice, many zones contain only one domain–this is the case for top level domains as well as large domains in general. In the rest of this article, we use the terms domain and zone interchangeably when a zone contains a single domain.

Every RRset in the DNS belongs to a specific zone and is stored at the nameservers of that zone. For example, the RRset for ⟨`www.ucla.edu, IN, A`⟩ belongs to the `ucla.edu` zone and stored in the `ucla.edu` nameservers. Two important types of RRs, the NS RRs which hold the names of DNS servers, and the corresponding A RRs which hold the IP addresses of the NDS servers (called "glue records"), play a critical role in establishing and maintaining the DNS hierarchy. The NS RRset of each zone Z is stored both locally *and* at the parent zone P, so that the parent zone can refer the queries for Z's DNS names to Z's DNS servers. When a zone changes any of its DNS servers, it must notify its parent to update the NS RRset and A RRset stored at the parent zone.

End users and applications resolve a DNS name by querying the DNS for the corresponding RRset. Typically, a simple stub resolver is implemented on every host which sends DNS queries to a local *caching resolver* which takes the responsibility of walking the DNS hierarchy to get the final answer and then sends the answer back to the stub resolver.

B. DNS Security Extensions

DNS was designed without security as a central concern, and a variety of possible attacks against DNS have been identified [6]. An attacker can exploit these vulnerabilities to inject spoofed DNS data and re-direct user traffic (e.g., Web browsing) to incorrect and often malicious sites, leading to various denial of service and/or security breaches. A detailed threat analysis for DNS can be found in [5]. To defend against these threats, the DNS Security Extensions (DNSSEC) is designed to achieve two security goals: *data integrity* and *origin authenticity*. DNSSEC uses public-key cryptography (RSA for example) to enable each zone to prove the authenticity and integrity of its DNS data. To do so, each zone creates a public-private key pair, stores the public key in a new RR type called DNSKEY RR, signs its data (in units of RRsets) using the private key, and stores the signatures at the authoritative servers in another new type of RRs, called RRSIG. Whenever a DNSSEC-enabled server returns an RRset, it also returns the companion signature. A resolver uses the zone's public key to verify whether a received RRset matches the signature; a match indicates that the RRset was indeed originated from that zone and was not altered in transit. To resist replay attacks, each signature carries an expiration time, specified by a *definitive* timestamp, and becomes invalid beyond this timestamp. Accordingly, the cache discards an RRset when *either* its TTL *or* the companion signature expires, whichever comes first.

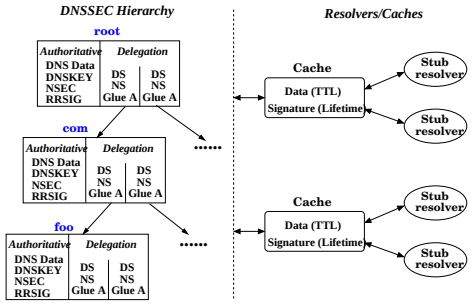


Fig. 1. An illustration of the DNSSEC system.

To verify the signature of an RRset, a resolver can query a zone for its DNSKEY RRset. To verify the keys, DNSSEC leverages the existing DNS delegation hierarchy to provide a Public-Key Infrastructure (PKI). In this PKI, each parent zone signs its children zones' DNSKEY RRs[1], while the public key of the root zone is distributed to all resolvers in a secure, out-of-band mechanism[2]. As such, starting from a root zone's public key, resolvers should be able to walk down the DNS hierarchy and verifying the public keys for each zone along the way. For example the *root* public key is used to authenticate the *org* public key, which in turn is used to authenticate the *foo.org* public key, and so on.

In addition to authenticating RRsets through RRSIGs, a zone must also provide authenticated answers when it receives queries for RRsets that do not exist. Because of the desire to keep the private keys offline, upon receiving the query for a non-existing name, a zone cannot sign a denial of existence response in real time. Instead, authentication of denial of existence is achieved in the following way. A zone first sorts all the existing names in a canonical order, then creates an RR of a new type, called NSEC, for each of its names, and signs

[1]The exact mechanism is slightly more complex: the parent zone stores and signs a DS RR that is a hash of one of the child's DNSKEY RRs. Section III-A discusses DS RRs in more detail.

[2]Although this approach looks similar to that used in distributing the DNS root servers' IP addresses, one fundamental difference here is that cryptographic keys need to be changed periodically, however long the period may be, while the IP addresses for root servers do not have this requirement. Rolling over the root public keys requires updating *all* DNS resolvers on Internet; how to do it effectively remains an open issue.

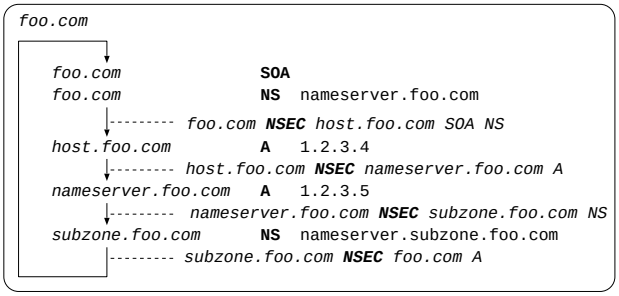


Fig. 2. An illustration of NSEC RR usage.

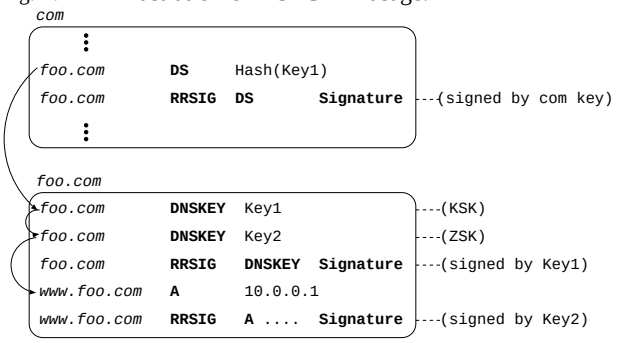


Fig. 3. One level of indirection is added in the trust chain through DS RR, KSK, and ZSK.

these NSEC RRs using the private key. The data portion of an NSEC RR indicates which RRsets exist under the name and identifies the *next* existent name in the zone. An NSEC RR, together with its signature, can prove the non-existence of the queried data. In the example of Figure 2, four distinct names exist in the zone *foo.com*, and four NSEC RRs are created. The NSEC RR at *host.foo.com* indicates that only an A RR exists under this name, so it can be used to prove the non-existence of an NS RR under *host.foo.com*. It also shows that *nameserver.foo.com* is the next name after *host.foo.com*, which proves that the name *mailserver.foo.com* does not exist.

From a cryptographic standpoint, DNSSEC aims at a rather moderate goal, the design should be simple, and the deployment should not be difficult either. However the reality shows otherwise. The DNSSEC development efforts started in mid-1990's, and it took more than a decade and three rounds of revisions to have the DNSSEC specifications finalized in March 2005 [2], [4], [3]. Although some pilot DNSSEC deployments are underway, e.g., efforts by VeriSign, `.org`, `.se` and `.br` registrars, a number of issues remain open. The goal of our study is to understand why it is so difficult to add simple cryptographic protections to DNS, and what the fundamental challenges are.

III. Design Issues With A Hierarchical PKI

As we described in the previous section, DNSSEC builds a Public-Key Infrastructure (PKI) by leveraging the existing DNS delegation hierarchy. Specifically, each zone signs the public keys of all its children zones. However this seemingly simple PKI design ran into unforeseen problems when it was implemented and started to be deployed. In this section we show four specific examples of the problems encountered in practice: scalability, handling data that crosses administrative boundaries, coordination across these boundaries, and incremental deployability.

A. Scaling

DNSSEC authenticates a zone's public key by verifying a corresponding signature from its parent zone. Throughout the rest of this paper, we refer to this signature simply as the *parent's signature*. The choice of where to store this signature has no impact on its cryptographic security. Following the convention of storing the data and its associated signatures in the same place, the first DNSSEC specifications [9], [8] store both a zone's public key and its parent's signature in the (child) zone.

However this straightforward design decision overlooked one important factor: a DNS zone may have a large number of child zones. For example the *.com* zone has tens of millions of delegated children zones. With this design, whenever a zone changes its key, it must contact *each* of its child zones to update the signatures for their public keys, a process that is *operationally* infeasible for large delegation-centric zones, such as *.com*, *.org*, *.net*, or *.edu*. In addition, a zone must contact its parent to get an updated signature every time it changes its public key, and the frequency of such update requests at the parent zone also goes up linearly with the number of children zones.

Subsequent DNSSEC specifications made two changes to address the above mentioned scaling concerns. First, a new Delegation Signer (DS) RR is defined to store the hash of a child's public key *at the parent zone* and is signed by the parent key[3]. Thus whenever a zone changes its key, it can re-sign all its children's DS RRs stored *locally*, without notifying any of the children zones. A child zone stores its public key in a DNSKEY RR, which is verified if the corresponding DS RR's signature can be verified by the parent's public key.

The second change is to allow a zone to have *multiple* public keys; one is called the Key Signing Key or KSK (because its only job is to sign other keys) and the rest are called the Zone Signing Keys or ZSKs (because their job is to sign zone data), as shown in Figure 3. The hash of the public part of KSK is stored as a DS RR at the parent zone. The private part of KSK is used to sign the DNSKEY RRset (which includes both the KSK and the ZSKs). The ZSKs are used to sign the zone's data records, and these signatures are verified by first verifying the zone's KSK, which is then used to verify the signature of DNSKEY RRset that contains the ZSKs, and finally the ZSKs are used to verify the signature of the data. With this separation of KSK from ZSK, a zone can change any of its ZSKs locally, and contacts its parent to update the DS RR only when its KSK changes. Because the KSK is used to sign the DNSKEY RRset only, it can also be better protected (e.g., kept offline), and its vulnerability to cryptanalysis attacks is reduced. For example, [13] recommends that a zone should change its ZSKs once per month and its KSK once per year.

[3]The main motivation for storing a hash of a child's public key, instead of the key itself, is to save the storage overhead.

From a cryptographic standpoint, both the original DNSSEC specification, as documented in [9], [8], and the current revision with the above two changes (i.e. storing a child's DS RR and its signature at parent zone and the separation of KSK and ZSKs) give the same level of cryptographic protection. From a system viewpoint, however, the two later changes brought significant scalability improvement over the original design and made it feasible to deploy.

B. Boundary Crossing: Unsigned DNS Data

In an ideal cryptographic design, all data should be signed to assure authenticity and integrity. When applying cryptographic protection to a distributed system, however, new issues arise due to disjoint ownership of the data.

Because DNS is a distributed database that is managed by different administrative domains, it provides a strict definition of data ownership. Each RRset belongs to one, and only one, zone, and each zone signs all its authoritative data, including the DS RRs from all its children zones. However, as we described in Section II, the NS RRs and glue A RRs of each zone are stored in *both* the local zone and the parent zone so that DNS queries can perform a top-down lookup to find the intended server. Since the NS and glue RRsets are defined to be the child zone's authoritative data, the DNSSEC design left them *unsigned* in the parent zone. It is believed that the authentication chain should provide sufficient protection on the delegation, thus leaving the NS and glue RRsets unsigned at the parent zone causes no harm. For example, even if a man-in-the-middle can forge a false delegation NS RR to mis-direct DNS queries to a host of the attacker's choice, the attacker still cannot forge any DNS data in the child zone, because he does not have the child's private key to produce a verifiable signature for any false data.

However, a thorough analysis shows otherwise. Consider a scenario where an attacker obtained the private key of the zone *foo.com* but did not gain the control over the zone's DNS server: the attacker can forge a referral from *.com* and re-direct the queries for *foo.com* to a host of his choice. The forged answers will be accepted by DNS resolvers as the attacker knows *foo.com*'s private key. Had the *.com* zone signed the delegation NS RRs, the attacker would not have been able to forge a referral to re-direct the queries in the first place. This example shows that letting the parent zone sign its children's delegation NS RRs can bring additional protection in case a child's private key is exposed. This signing can be readily implemented with today's DNS practices, since a child zone is required to notify its parent of any changes in its name servers (through mostly a manual process).

Cryptographic design decisions require thorough analysis and good judgment to ensure both that all data gets protected and that the original system constraints are not violated. In the specific case of DNS delegation records, we believe the judgment call should fall on the side of protecting critical infrastructure records rather than mechanically adhering to the data ownership rule. Fixing this flaw requires revising the DNSSEC signing and verification rules to include the parent-side NS and glue records. Although the data is owned by the child zone, the handling is similar to that of DS RRs.

C. Cross-domain Coordination

As we discussed earlier, DNS maintains its name delegation chains through coordinating NS and glue A records between parent and child zones. Any changes made to these records by a child zone must be promptly propagated to the parent. However, because this coordination is done by human operators and human actions are prone to errors, a recent measurement study [26] shows that at least 15% of the zones suffer from configuration errors across the child and parent zone boundaries. Nevertheless, DNS can tolerate such imperfect parent-child coordination through the redundancy in DNS servers, and DNS name resolutions can succeed as long as the parent knows at least one correct authoritative server used by the child zone.

The cross-domain coordination required by DNSSEC is more demanding, because each parent and child pair need to coordinate not only DNS server changes but also periodic key changes. A more fundamental difference in the coordination is that DNS name resolution tolerates inconsistencies in NS and glue A RRsets between the parent and child zones. The delegation link works as long as the parent knows at least one correct nameserver of the child, and the chance of success increases when a child zone has more DNS servers. In contrast, DNSSEC *fails* whenever an attacker breaks one public key that matches the DS RR, thus having multiple DS RRs may decrease the strength of cryptographic protection, rather than increasing it.

There is no reason to believe that the DNSSEC deployment will do a better job in keeping the DS RRs updated than DNS has done in keeping NS and glue A RRs updated. Providing operational guidelines may help reduce such errors in cross-domain coordination, but human errors, especially in a global scale system, are inevitable. Recently there have already been evidences [1] that the uptake in DNSSEC adoption has included an increase in DS-related misconfigurations. A number of DNS monitoring tools and services have emerged to check cross-domain configuration errors [25], and similar tools such as [12] have begun to prove themselves as an integral component in the DNSSEC deployment.

D. Incremental Deployability

At first glance, it seems a natural choice to leverage the existing DNS delegation hierarchy to build a PKI for DNSSEC. Unfortunately this choice overlooks a fundamental constraint in deploying any new function in a decentralized operational system: the decision to deploy a new functions lies in the hands of individual DNS domains, thus it can only be rolled out incrementally over time, if it gets deployed at all. This reality runs counter to the DNSSEC design because an authentication chain from the root to a given zone Z can only be formed when *every* ancestor of Z has deployed DNSSEC. Individual zones cannot benefit from DNSSEC unless and until all of their ancestors have also deployed DNSSEC.

Given DNS is a distributed system maintained by millions of autonomous administrative domains, when individual domains make independent decisions to turn on DNSSEC, the result is multiple isolated *islands of security*. An island of security is

a subtree in the DNS hierarchy in which DNSSEC has been deployed. The public key for the root of this subtree is called a *trust anchor* [14]. Since the trust anchor KSK cannot be verified by its parent zone which has not deployed DNSSEC, other means are needed for resolvers to collect, verify, and maintain the trust anchor KSKs. Unfortunately, the DNSSEC design does not provide a mechanism for a resolver to obtain the KSKs from a large number of DNSSEC islands in a secure and scalable manner.

As DNSSEC is being slowly rolled out, a set of operational guidelines have been developed. The current guidelines suggest that each caching resolver be manually configured with the trust anchor for each isolated DNSSEC island. Although this manual configuration approach can work in a small scale, its feasibility decreases as the size of the deployment base increases. Because each caching resolver has to update its configuration file whenever a trust anchor KSK changes, when the deployment base grows, this approach will suffer from the scalability problems in both the number of DNSSEC-capable caching resolvers and the number of trust anchors.

A few proposals have been made to overcome the difficulty of manually configuring all the trust anchors in all DNSSEC-enabled caching resolvers [15], [17]. The basic idea is to inter-connect the otherwise isolated DNSSEC islands through cross-signing, in which the roots of different islands can sign each other's public keys to form a web of trust, similar to the PGP model. However specifics and security analysis of these proposals are missing. A more recent proposal, DNSSEC Lookaside Verification (DLV), suggests to have a zone's public key signed by trusted authorities outside the DNSSEC system, e.g., VeriSign or ISC. Overall, DNSSEC's lack of provisioning for incremental deployment has seriously hindered its deployment, and all the attempts so far to retrofit various patches into the original design are yet to be proven effective. We believe that an effective solution to DNSSEC's incremental deployment can be taken from a distributed monitoring framework. In Section VII we outline a proposal that incorporates a number of advantages of other approaches, and overcomes a number of their drawbacks as well.

IV. Impact of DNS Caches

Caching is a fundamental part of the DNS and affects both the design and operation of DNSSEC. As discussed in Section II, end hosts implement a minimal *stub resolver* that directs queries to a caching resolver. The caching resolver handles all the complex functions of traversing the DNS hierarchy, obtaining the requested data from authoritative servers, and returning the responses to the stub resolver. Note that stub resolvers directly communicate with caches, but only indirectly communicate with authoritative servers.

To understand the impact of caches, consider a problem where Alice (the stub resolver) wants to authenticate data from Bob (the nameserver), despite the actions of malicious player Eve. However, Alice cannot directly communicate with Bob. Instead Alice can only send messages to a fourth player, Carol, who represents the cache. Carol may answer the question using cached data that *Carol believes to be correct* or may contact Bob to obtain the answer.

Carol's actions and policies have direct impact on Bob (the nameserver). During normal operations, Bob may update data and refresh signatures, change public keys, revoke public keys, and so forth. But any change made by Bob is not immediately visible to Alice. Alice may continue to receive cached data from Carol long after Bob has replaced the data or the keys. Sections IV-A and IV-B discuss how caching affects Bob's approach to key rollover and key revocation.

Carol's actions and policies also have direct impact on Alice (the stub resolver). If Carol fails to apply adequate security policies, Eve can poison Carol's cache with false data. Alice may detect the cached data is invalid, but Alice does not communicate directly with Bob to obtain the correct data. Carol can also introduce problems by incorrectly rejecting valid data. For example, an incorrect clock may cause Carol to believe valid signatures have expired. In this case, Bob has correctly signed a message and Alice could correctly authenticate it, but Carol rejects the message and prevents it from reaching Alice. Section IV-C examines such interplay between the security policies at Alice (the stub resolver) and Carol (the caching resolver).

Finally, the lifetime of cached data depends on a relative Time To Live (TTL) value and fixed signature lifetime. Each answer from Bob includes a TTL and Carol may cache for the data for the next TTL seconds. Each signature from Bob also includes a fixed expiration time and Carol should discard the data after the expiration date. Section IV-D examines how the TTL and expiration time interact; some combinations can lead to update explosions at Bob.

A. Key Rollover

Best operational practices state that zones should not use the same key pairs forever, and that the keys will need to change over time. The objective of key rollover is to phase out an old key and replace it with a new key, as part of routine operational practices. Section III-A described the parent-child coordination that is needed to admit the new key. However, deleting the old DNSKEY RR (or DS RR in cases of KSK rollover) from the nameservers does not remove it from the system. It may continue to exist in the caches around the Internet, and the old signatures generated by this key may have also been cached. These cached entries are deleted only after their TTLs or signatures expire.

Ignoring the effect of caching can break the authentication chain. For example, a stub resolver may receive an old signature from cache and query for the key to verify the signature. If the caching resolver does not have the key and the authoritative server replies with a new key, then the stub resolver cannot authenticate the cached data and hence has to reject it.

Operational guidelines [13] introduce a grace period for the old key during the key rollover process. Consider the scenario where a zone changes its KSK. The zone first adds the new KSK to its DNSKEY RRset, but keeps the old KSK to preserve the authentication chain. Eventually, the old DNSKEY RRsets will time out (e.g., after one TTL), and every cache will either have no DNSKEY RRset or have a DNSKEY RRset with both

the old and the new KSK. At this point, the DS RR at the parent is changed to match the new KSK. The authentication chain is still preserved because any cache with the old DS RR will match the old KSK and any cache with the new DS RR will match the new KSK. Again the zone waits until all cached DS RRs expire. At this point, all caches either have no DS RR or have the new DS RR. The old KSK can now be removed from the DNSKEY RRset, and the new authentication chain through the new KSK/DS RR starts to take effect[4].

A similar approach [30] uses multiple keys and multiple DS records. In this approach, a zone can publish both an *active* KSK and a *standby* KSK. Both KSKs have corresponding DS records stored at the parent zone. In a key rollover, the active KSK is retired, the stand-by KSK becomes active, and a new KSK is added as the new standby KSK.

In addition to the intricacies of changing the zone keys, operators also need to consider the possible need to evolve the cryptographic *algorithms* in use, e.g., from RSA to GOST. The processes governing such cryptographic algorithm rollover are still not fully specified, but they will likely follow similar steps used for key rollover.

B. Key Revocation

The single key rollover process from [13] (described above) assumes that the old key is still secure during the rollover period. It also assumes that the TTL value is honored and an attacker is not actively trying to replay old information. However in case of key compromise incidents (or suspected compromise), referred to as "emergency key rollover" by [13], prompt *key revocation* actions are needed. During the key revocation period, the old private key is presumably known by the attacker and may be used to forge records in the zone; the attacker is also not constrained by the TTL in replaying the old (compromised) public key. In fact, the attacker can continue to replay the old key until the definitive expiration time of the associated signature (e.g., the signature on the DS RR for a KSK, or the signature on the DNSKEY RRset for a ZSK) expires, which may take a few days or even a month [13]. Until then, DNSSEC authentication in the zone and all its descendant zones is essentially compromised. Ideally, one would like to revoke the compromised key as soon as possible.

The multiple key rollover process from [30] uses both active and standby keys. If only the active key is compromised, one can revoke that key and immediately switch to the standby key. Similarly, if only the standby key is compromised, one can immediately revoke the standby key.

Unfortunately, the DNSSEC specifications did not provided an explicit key revocation mechanism until the publication of [30] in late 2007, and currently the choice to implement this new specification is left to individual zone operators[5]. In the absence of universal key revocation support, regardless of what actions the zone operators take, an attacker can always replay the compromised key and use it to successfully forge DNS records until its signature lifetime expires. It is stated in [13] that "zone operators have to make a tradeoff if the abuse of the compromised key is worse than having data in caches that cannot be validated".

C. Cache-Stub Verification Policies

In the current DNS, caching resolvers handle all the complexities of traversing the DNS hierarchy and obtaining any requested data. With the transition from DNS to DNSSEC, a natural choice is for caching resolvers to handle the complexities of building an authentication chain and verifying all received data. However this approach presents a security concern if the stub resolver does not trust the caching server. For example, a user accessing the Internet from a hotel may use a caching resolver offered by the hotel network. Although the caching resolver may not be malicious, the user does not know whether the caching resolver is configured with desired trust anchors and security policies. Different configurations may result in verification conflicts, namely *false negatives* when the caching resolver rejects answers that the stub resolver considers valid, or *false positives* that the caching resolver returns answers that would have been rejected by the stub resolver.

To reduce the stub resolver's dependency on the caching resolver, DNSSEC allows a stub resolver to enforce its own local policy through a Check Disabled (CD) bit in the query. When a caching resolver receives a query with the CD bit on, it should forward answers to the stub resolver without performing verification. As such, the CD bit allows a stub resolver to use the caching resolver solely as a DNS cache, rather than a DNSSEC verifier. This addresses false negatives, but false positives are more difficult to overcome. There is currently no mechanism for a stub resolver to flush the cache at a caching resolver. Once a caching resolver accepts an answer as valid, that answer is entered in the cache. Even though the stub resolver may consider the answer invalid, the caching resolver will continue to return the same answer until the TTL expires. The only way for the stub to obtain a different answer is to use another caching resolver or directly resolve the query itself.

DNS heavily relies on caching to reduce server load and improve query resolving performance, with an *implicit* assumption that stub resolvers trust whichever caching resolvers they may default to. DNSSEC exposes this trust relationship issue between stub and caching resolvers. Instead of using a default caching resolver by the Internet connectivity provider, as in the current practice, stub resolvers must now make an explicit decision of either performing verification on its own, which defeats DNS caching, or being configured to use a set of trusted caching resolvers only.

D. Cache Synchronization

DNS caches rely on the *relative* TTL value in a RRset to decide when to remove this RRset from cache. On the other hand, DNSSEC puts a *definitive* expiration timestamp in each signature, beyond which the signature becomes invalid. To support both TTL and signature lifetime, DNSSEC modifies

[4]ChangING a ZSK does not involve the parent zone, but similar reasoning holds with respect to caches.

[5]Some notable TLDs, such as .com, have asserted that they do not intend to implement RFC 5011.

the DNS caching rules as follows. A cache should discard a RRset as soon as *either* its TTL expires *or* the companion signature expires, which ever comes first.

However, this seemingly reasonable change can lead to synchronized actions among caching resolvers. When a RRset's signature expires, all the caches that hold this RRset discard it exactly at the same time. If the record is popular and frequently queried, such as the case for *cnn.com* or *google.com*, these caching resolvers are likely to fetch the RRset again more or less simultaneously, leading to a query implosion at the authoritative servers. We call this phenomena caused primarily by the definitive signature expiration time the *cache-sync* effect. Note that TTL expiration does not lead to this problem because the fetching time of popular DNS names by different caching resolves are likely different, thus their TTL expiration time differs as well.

A DNSSEC operator is likely to sign and re-sign an entire zone at the same time to minimize the operational overhead. Consequently all the DNSSEC signatures are likely to be assigned the same expiration time. This can further exacerbate the cache-sync effect since all RRSIG RRsets of the zone will expire simultaneously, leading to an instant high volume of queries to the zone's authoritative nameservers.

To quantify the impact of cache-sync effects, we develop a simple analysis that considers a single *A* RR served by one authoritative server and fetched by multiple caching resolvers. The queries are sent to each cache following a Poisson process with an arrival rate of λ, which represents the popularity of the RR. We first analyze the average load at the authoritative server. Because signature life time is typically orders of magnitude longer than query inter-arrival time [13], we neglect the signature lifetime and consider only the TTLs. It can be easily seen that each cache sends queries at an average rate of $\frac{\lambda}{1+\lambda \cdot TTL}$. Next we analyze the peak load at the authoritative server by taking into account DNSSEC signature expiration. When the signature expires, both the A RR and its RRSIG are immediately deleted from all caches, and the next query causes the cache to request the A RR from the authoritative server. The peak outgoing query rate per cache is equal to the incoming query rate, i.e., λ. The impact of cache-sync effects can be shown by the ratio between peak and average load as:

$$\gamma = \frac{\lambda}{\frac{\lambda}{1+\lambda \cdot TTL}} = 1 + \lambda \cdot TTL \qquad (1)$$

For example, if we assume the TTL value is 1 day and the query rate (λ) is one per minute, the peak load would be 1380 times of the average. The cache-sync effect becomes more pronounced with globally popular RRs, i.e., when λ increases, and with larger TTL values, because a large TTL lowers the server load by more effective caching, but fails to suppress the queries upon signature expiration.

Such unintended synchronization behavior is not unique to DNSSEC signature lifetime. A naive reliable multicast design using either ACK or NAK can trigger an ACK (or NAK) implosion at the source upon a successful delivery (or a packet loss). It is well-known among the protocol design community that protocol designs must avoid triggering synchronized actions in large-scale distributed systems. We hope this guideline will be followed by future cryptographic designs as well.

Avoiding Cache Synchronization Recall that the caches are synchronized when they simultaneously delete a RRset upon its signature expiration. One can reduce cache-sync by making the TTL of a RRset expires before the companion signature. The DNSSEC operational guideline [13] requires the zone operator to replace all signatures at least one TTL before their expiration time. In practice, however, the cache synchronization problem still exists. First, the zone operators may ignore this guideline or accidentally forget to update the signatures in time. Secondly, even when the operators carefully follow the guideline, cache-sync effects may still occur, e.g., when the authoritative servers are configured with wrong clocks, or a network partitioning prevents a secondary server from performing zone transfer from the master server.

A second line of defense against cache synchronization is to perform *TTL trimming* at the caching resolver. Specifically, when a caching resolver receives a RRset, it checks whether the TTL or the signature expires first.Let T_r denote the time the RRset is received and T_e denote the signature expiration time. If $T_r + TTL > T_e$ (i.e., the signature expires first), the caching resolver trims the TTL into a random value in the range $[0, T_e - T_r]$. This makes TTL expires at a random time within the signature lifetime.

V. Impact of Heterogeneous Operations

Another important issue in DNSSEC operations is whether a zone should keep its private keys online or offline. A DNSSEC zone must ensure the secrecy of its private keys yet sign all its authoritative RRsets using these keys. To this end, the foremost operational concern is where to store the private keys and which entity has access to them. In this section, we first identify the fundamental conflict between the need to keep keys *online*, as required by signing dynamically generated or changed data, and the desire to keep keys *offline* for better protection. We then examine two open issues resulted from this conflict, namely authenticated denial of existence and secure dynamic updates.

Ideally, one would prefer the offline key approach to better protect the secrecy of the keys. In this approach, a zone stores its private keys offline, e.g., in a non-networked and physically secured computer called a *signer*. When the zone data needs to be signed, the master server sends the zone data file to the signer, which signs it using the private key, and sends resulting signed zone file back to the master server. The master server then sends this updated signed data to all the secondary servers. Because the private keys are not accessible online, the chance of their exposure is greatly reduced. For this reason, the DNSSEC specification [4] recommends the practice of using offline keys. However, offline keys make it difficult to sign dynamic updates, it is infeasible to invoke the offline signer every time a piece of new data needs to be signed. We will further elaborate on this issue in Section V-A.

On the other hand, although keeping the private keys online makes signing dynamic updates easy, it exposes the keys to greater security threat in DNSSEC operations for two

reasons. First and foremost, putting the private key online at multiple DNS servers may impose high security risks. In global DNS, each zone deploys redundant authoritative servers, also called *secondary servers*. To maximize service reliability, it is recommended that the redundant servers be placed in topologically different locations, i.e., locations that are managed by administrators other than the zone's owner. Therefore, keeping the keys online opens a large margin for errors. Second, the compromise of a single private key may incur a domino effect. In DNSSEC, the private key of a zone is used not only to sign the DNS data in the zone but also to sign the delegations of all the child zones. The compromise of one zone's key may cause chain effects leading to the compromise of multiple DNS zones.

A. Handling Dynamic Updates

Dynamic DNS has become a popular practice due to the widespread use of DHCP. When a host is allocated a new IP address by DHCP, it needs to update its *A* record. DNS has adopted an automatic update mechanism which greatly reduces the administrative burden in accommodating frequent DNS data changes. In the future, dynamic DNS may also be used to handle IP address changes caused by host mobility. When a host moves to a new location and obtains a new IP address, it can use the dynamic DNS update mechanism to change its *A* record.

A dynamic update may change several records in the zone, including the updated RRset itself, the NSEC RRs under this name and adjacent names, and the SOA RR[6]. All these new records must be promptly signed. However, if the zone key is kept offline, the DNS operator must be involved in invoking the offline signer, which defeats the purpose of having an automatic update mechanism in place. The current DNSSEC specifications (see RFC 3833) acknowledges this conflict: "a zone-signing key must be available to create signed RRsets to place in the updated zone. The fact that this key must be online (or at least available) is a potential security risk." Note that the resolution of this conflict is not a theoretical but an operational issue, and only careful engineering can lead to a sound solution in practice.

B. Authenticating Denial of Existence

The second challenge resulting from the different practices of online and offline keys is authenticated denial of existence, which requires DNSSEC to provide authenticated answers to queries asking for nonexistent records. The online key approach offers a straightforward solution to the problem. When DNS answers with a "non-existent record" reply to a query, the server can use the online key to construct and sign the non-existence proof, but at the cost of keeping a zone's private keys online at *all* its name servers.

To avoid putting private keys online, the zone must construct and sign the proofs of non-existence *a priori*. The NSEC scheme in the latest DNSSEC specifications [3] (described in Section II) takes this approach. Unfortunately, it suffers from the so-called "zone walking" problem, as one can easily retrieve the complete list of records in a zone in the following way. Consider an example of *foo.com* zone. One first queries for the *foo.com* NSEC RR, and the answer reveals all record types that are present at name *foo.com*, as well as the next name after *foo.com*. Suppose that the *foo.com* NSEC record lists *a.foo.com* as the next name. One then queries for the *a.foo.com* NSEC RR to learn all types at *a.foo.com* and the next name after *a.foo.com*. By repeating the same step until reaching the end of the zone, one retrieves the entire zone data in the number of steps equal to the distinct names in the zone. In addition to zone walking, the NSEC scheme also incurs high cost as the addition of NSEC RRs roughly doubles the zone file size, even when only a few RRs are signed at the beginning. This initial overhead is particularly troublesome for large delegation zones, such as .com, with millions of records.

A number of DNS operators raised concerns on this exposure of privacy introduced by NSEC. In fact, the different views on NSEC RR's legitimacy created a road block in DNSSEC development for a while. People who did not view zone walking as an issue argued that, as an open database, DNS does not, nor should it attempt to, protect the privacy of data in a zone. However others felt strongly that, compared to the current DNS, NSEC made it much easier to obtain the entire zone data, which should be prevented for security and legal reasons. For example, a complete zone file may be used by the spammers as a source of probable e-mail addresses, or by the scanners to infer the internal network topology and services. Together with WHOIS queries, the complete zone data can be used to reveal registrant data, the information that may be protected under the law (e.g., in Europe).

The debate over NSEC ended with the recognition that, to move DNSSEC forward, the design must accommodate different requirements. One proposal to remove the privacy concern is the minimally covered NSEC [32] scheme which assumes the private key is online. When a server receives a query for a non-existing name (*QName*), it generates and signs a minimally covering NSEC RR. The *owner* and *next* names in this NSEC RR are $QName-\epsilon_1$ and $QName+\epsilon_2$, respectively, where ϵ_1 and ϵ_2 are two small, randomly chosen values such that no existent names fall into this range[7].

More recently NSEC3 [16] was proposed as another solution to avoid zone walking without keeping private keys online. It hashes all existent names and sorts these hash values, as opposed to the original names, as a chain. It generates an NSEC RR for each hop in the hash value chain, i.e, the *owner* and *next* names in each NSEC RR are two adjacent hash values. Thus, a signed NSEC RR in this design proves that no name exists whose hash value falls into the specified range. While it is still subject to dictionary attacks, the use of hash chains, instead of name chains, makes zone walking much more difficult. On the other hand, NSEC3 (as well as NSEC) introduces significant protocol complexity, in terms of both zone signing and response validation, due to the use of

[6]A dynamic update may trigger a change to the zone serial number, which is a part of the zone's SOA RR.

[7]With online keys, a simpler design would be to directly sign the non-existence replies. However, the minimally covered NSEC was proposed for compatibility with the NSEC scheme in the DNSSEC specifications [3].

wildcards and the existence of unsigned delegations. It also increases the size of the zone file because one NSEC3 record and an accompanying signature are created for each name.

Perhaps one of the most direct concerns with NSEC3 is also one of the most unforeseen: Path Maximum Transmission Unit (PMTU) limitations. Prior work [24] has illustrated that oversized DNSKEY messages can exacerbate PMTU limitations and lead to availability problems for zones. One of the unforeseen side effects of NSEC3's design is that its messages can be as large as, and in some cases larger than, DNSKEY messages. Thus, a zone that is trying to prove the denial of existence for a name may generate a NSEC3 reply that cannot be received by the querying resolver.

VI. DNSSEC Monitoring and Deployment Status

In this section, we use the results from the first DNSSEC monitoring system, called SecSpider [12], to review the preceding discussions in the context of actual DNSSEC deployment. SecSpider is a distributed monitoring system that issues the same DNSSEC queries simultaneously from multiple polling locations (or pollers) distributed around the Internet. The goal of SecSpider is to monitor certain important facets of zones to test them for DNSSEC RFC compliance, to check the zones' operational statuses, and to observe the served data from multiple diverse locations over time. By tracking DNSSEC-specific records (e.g., DNSKEY sets) from multiple locations and over time, SecSpider can detect Man-in-the-Middle attacks that resolvers may fear. At the time of this writing, SecSpider has installed pollers in America, Europe, and Asia and is in the continuous process of adding more locations.

As we mentioned previously, DNSSEC has been undergoing its rollout phase for the last several years. Yet, it is still in its nascent stages. Based on the latest monitoring results from SecSpider, we estimate that there are only about 10,523 DNSSEC enabled zones operating in the wild. Furthermore, the corpus of zones monitored includes both production zones that represent operational entities who serve production data and have opted to augment their DNS zones with DNSSEC protection, *and* test zones who have rolled out DNSSEC in some form of test capacity. Testing zones may have any number of agendas for deploying DNSSEC, but in our loose classification we attempt to identify and separate their behavior from those zones which appear to be more operationally attended to.

We perform a very simple test to determine if a zone is a production zone or a test zone. This test is in no way conclusive, but is used as a low-pass filter to get a general sense of which zones operate in which capacity. For each DNSSEC (or secure) zone in our corpus, we presume any Top-Level Domain (such as `.se`, `.br`, `.org`, etc.) or any zone under the `.arpa` TLD to be defined as a production zone. For all other zones, we query to see if there is an active webserver at a `www` record *or* if there is an `MX` (or mail exchanger) record that corresponds to an active mail server. If either of these is true for a zone we broadly classify it as a production zone.

As an example, this simple, automated, litmus test successfully distinguishes test zones like:

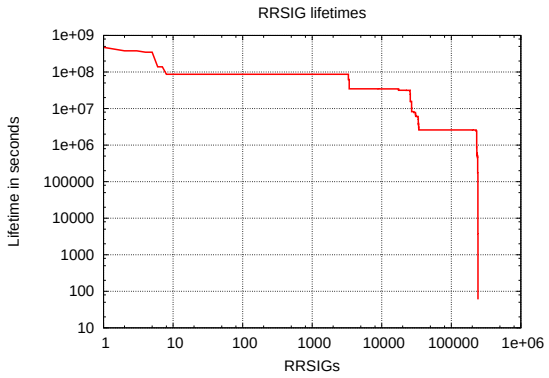


Fig. 4. This Figure is drawn in log-log scale and shows the distribution of the RRSIG lifetimes of RRsets in the production zones.

`bogussig...test.jelte.nlnetlabs.nl.`

from production zones like `nanog.org.`. Indeed, many of the test zones undertake operational practices that would be ill-advised for production zones. For example, the distribution of RRSIG lifetimes for non-production differs from most production zones. The RRSIG lifetime distribution of production zones can be seen in Figure 4. Previous work has observed that production zones and test zones behave very differently, and separating out the latter from the former can allow for much more precise analyses [24]. Based on this, each of the following analyses are done using production zones only.

Based on the above classification, we currently estimate that there are roughly 1000 production DNSSEC zones at the time of this writing. While the size of DNSSEC deployment is still quite small, many of the operational complexities may start to reveal themselves to operators as the size grows.

The status of the DNSSEC hierarchy also indicates the immaturity of the deployment. Currently SecSpider tracks approximately 730 independent islands of security. The implication of this statement is that each of these islands is a secure zone whose parent has either not enabled DNSSEC, or who does not currently have a valid DS record that securely delegates to the island's root. One can see that this number represents 76.6% of the total number of secure zones. Moreover, 97.5% of these islands are of size 1 (i.e. with no secure parent or secure children). If a resolver were to manually configure a trust anchor for each island of security, it would have to manage rollovers, and churn for a list of 730 DNSKEY RRsets. While this number may be manageable today, as the deployment size grows this process could quickly become unrealistic.

In addition to the size of this trust anchor list, the administrative composition of these islands is interesting too. By examining the specific nameservers that serve each zone in an island we can generally estimate how many independent administrative authorities make up an island. In other words, by looking at which zones are served by the exact same nameservers, we can guess how many zones in an island belong to the same operational group, versus how many islands are actually composed of independent parties that represent an increasing adoption of DNSSEC. Figure 5 shows a selection of some of the larger DNSSEC islands and compares their

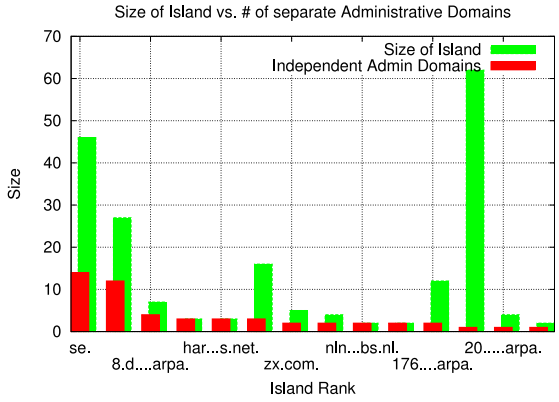


Fig. 5. This figure shows the size of several DNSSEC islands of security in contrast to the apparent number of independent administrative domains for each. Some islands of relatively large size appear to be run by a small number of independent administrative domains, while other islands are more diverse.

RRSIG lifetime	Average time of use of DNSKEY
0 - 30 days	83.20 days
31 - 60 days	209.19 days
$>$ 30 days	156.76 days

TABLE I
THIS TABLE SHOWS THE RRSIG LIFETIMES OF DNSKEY RRSETS VERSUS THE ACTUAL AMOUNT OF TIME THESE DNSKEYS WERE SERVED. MANY KEYS ARE RE-SIGNED MULTIPLE TIMES BEFORE BEING REPLACED.

sizes to the number of administrative domains that they are composed of. One can see that some large DNSSEC islands do not actually represent an increase in DNSSEC's adoption.

It is important to distinguish a key's lifetime from its signature's lifetime. More specifically, many DNSKEYs are re-signed multiple times before being rolled over. Operators may generate a key on one day, sign it every 30 days, and continue to use the same key for years. Table I shows a breakdown of observed keys lifetimes as compared to signature lifetimes of DNSKEY RRsets.

VII. DISCUSSION

So far we have scrutinized a number of operational issues as DNSSEC rolls out from a simple cryptographic design on paper to a deployed system in the Internet. We have also witnessed how these operational issues, which largely arise from the Internet's large scale and heterogeneity, have challenged, complicated, and sometimes even invalidated the DNSSEC protocol design. In this section, we seek to unify the otherwise piecewise analysis of individual problems and provide a *root cause* view on the challenges in deploying cryptographic solutions in large-scale systems. We will also discuss the importance of distributed monitoring and address two open issues raised in previous sections: incremental deployment and key revocation.

A. Why Is It So Difficult?

The aforementioned operational issues clearly reveal a fundamental gap between cryptographic designs and deployable solutions in Internet-scale systems, which is often overlooked. As the saying goes, "In theory, there is no difference between theory and practice. But, in practice, there is." [33] A cryptographic design is normally considered successfully completed when it is proven to meet the required cryptographic properties; specifics in its deployment are seen as having no impact on the cryptographic correctness. To real Internet systems, however, a cryptographically correct design is useless unless it is deployable. Deployability on the existing DNS requires the design to be scalable with large numbers of zones at any level, to enable incremental rollout, to effectively co-exist with DNS caching, and to tolerate the wide range of operational heterogeneity that necessarily exists in the global Internet.

One fundamental challenge in operating DNSSEC is that the system spans over tens of millions of independent administrative domains, while the provision of security, as defined in DNSSEC, requires symphonic actions from all of them. Such a cross-domain operational challenge manifests itself in several aspects in DNSSEC. First, the public keys of different zones are authenticated through a single hierarchical PKI. Any local change in a zone's public key may require synchronization across administrative domains, because the new key must be authenticated by the zone's parent and used to authenticate the zone's children. This causes serious scaling issues for the domains that may have millions of children zones underneath them. The coordination process is also lengthy and error-prone as it involves human operators, yet any out-of-sync configurations can break the chains of trust and disrupt the DNS service due to authentication failure.

The hierarchical PKI also presents challenges to incremental deployment, which is the only viable strategy to roll out any new functions in Internet. The DNSSEC design had assumed a systematic rollout from the top to the lower layers of the DNS name hierarchy, and thus did not provision for individual zones to turn on DNSSEC independently from their parents. But the latter is the reality, and during this time, the PKI is incomplete and there exist many isolated islands of DNSSEC-deployed zones, which cannot authenticate their public keys through the planned chains of trust, as each cannot demand its parent zone to turn on DNSSEC. As such, letting all the DNSSEC-aware resolvers securely acquire the public keys of all these islands becomes a new problem of its own.

In addition, caching is a fundamental part of the DNS that imposes a unique challenge to DNSSEC operations, setting DNSSEC apart from other cryptographic designs. Due to the existence of DNS caching, which often uses a long timeout value to improve DNS scalability, changes to the public keys of a zone will not be immediately visible to all resolvers. Rather, the old keys will continue to be used by caching resolvers all over the Internet until they run out their cache lifetime. Handling the co-existence periods of old and new keys leads to both the design complication and system vulnerability.

Lastly, zones in different administrative domains necessarily operate with different practices. For example, some zones may desire the privacy of their data due to legislative or security concerns, while others may not. Some zones may prefer to store the private keys offline for better protection, while others may choose to store them online at the name servers and protect the keys by other means, e.g., through the use of

specialized hardware. Cryptographic designers may either be unaware of such heterogeneity in practice, or fully appreciate the importance to accommodate the heterogeneity. That is why the original specification of DNSSEC dictated a uniform policy for all zones. It remains a design challenge to develop a cryptographic protection system that can simultaneously provide provable security while leaving the implementation policies and flexibilities to individual zones operators.

B. Importance of Distributed Monitoring

Our DNSSEC monitoring efforts have exposed a number of problems that only surfaced in actual deployments. First, data in Internet systems is not always universally available. Issues such transient failures, misconfigurations, and in particular roadblocks due to the existence of middleboxes are a fact of life for these systems. Second, cryptographic operations represent a new challenge on their own, lack of basic understanding and experience have led to additional errors besides misconfigurations. Furthermore, our results confirmed our analysis that the DNSSEC's design assumptions are not congruent with the common requirement that every party in the Internet tends to make their own decision about whether/when they may deploy new functions, or if deploy them at all; the result is a large number of isolated DNSSEC islands that simply does not scale.

Our monitoring results show that even in its early deployment stage, DNSSEC is a highly dynamic and a continuously evolving system. Thus, its behaviors must be continuously monitored to capture new failures and challenges. By measuring one gets data and that can inform a system's design, by quantifying data one can decipher its meaning and gauge the progress, and by monitoring one is able discover problems as they arise so that designs can be revisited.

Our monitoring system also inspired us to develop a practical and effective solution to DNSSEC incremental deployment problem, as we discuss next.

C. Facilitating Incremental Deployment

The incremental deployment of DNSSEC requires that, in the absence of a PKI, a caching resolver find the public key of each DNSSEC-enabled zone in a trustworthy way. We believe that a distributed monitoring system, such as *SecSpider* [12], can help fulfill this requirement. *SecSpider* has already made available the collected keys from all the known DNSSEC enabled zones. The remaining question is: how trustworthy are these results?

First, *SecSpider* is a distributed monitoring system, which makes it difficult for an adversary to compromise the results collected by each of all the monitors that are diversely located, assuming that the servers for each zone are also diversely located. If any mismatch between the replies received from different monitors is detected, instead of trying to determine which one is correct, the repository will provide the *complete* information to resolvers, allowing them to make an informed decision on which trust anchor to use, perhaps through consultation with additional information. An error will also be logged for the repository administrator to take actions.

Second, each DNSSEC enabled zone can register itself with SecSpider and periodically check the correctness of its own public keys as displayed on SecSpider, and promptly inform SecSpider of any discrepancies. We believe that each zone would have strong incentives to do such checking, which further increase the validity of SecSpider data.

Third, simple approaches exist to fetch SecSpider data in a secure way. SecSpider's public key is posted on its website. SecSpider will also watch its own key from multiple vantage points. Caching resolvers can get SecSpider's public key through either an offline secure channel, or simply grab it from SecSpider website, perhaps through multiple diverse lookup paths to minimize man-in-middle attacks. Caching resolvers can then query $SecSpider$ for the latest trust anchors of any DNSSEC zone, and the authenticity and integrity of $SecSpider$'s replies are ensured by its signatures.

The above proposed solution leverages the public nature of the SecSpider repository to achieve *security through publicity* [22]. Because data in $SecSpider$ is available to everyone as public knowledge, any errors are exposed to the affected zones when they check their own keys periodically. Upon detecting a trust anchor conflict, the zone owner can resolve it out-of-band with the repository administrator.

We note that the trust anchor repository is *not* meant to be a replacement to DNSSEC's PKI. Rather, its goal is to serve as a readily deployable, and a complimentary, solution to facilitate the incremental rollout of DNSSEC. When a parent zone enables DNSSEC, its children can have their public keys signed by the parent, and the parent may register with the repository as a means, external to the DNSSEC PKI, to advertise and authenticate its public key.

D. Addressing Key Revocation Problem

Whenever a zone's private key is compromised, it must *immediately* replace the compromised key with a new one to minimize any potential damage. The challenge is how to revoke the old keys from all the caching resolvers that still hold a copy of the old key. Problems may occur if a resolver tries to use the old (compromised) key to verify a signature generated by using the new key.

Two similar approaches have been proposed for explicitly marking public keys as revoked. [30] adds a *revoked flag* to the DNSKEY record itself; setting this bit to 1 indicates the public key has been revoked. [23] creates a new REVKEY record, similar to the DNSKEY record; if a public key appears in a REVKEY record then the public key has been revoked. Both mechanisms require self-signatures by the corresponding private key. The self-signature ensures that a public key can be revoked only by someone with access to the private key.

The revocation bit and REVKEY record allow an authoritative nameserver to declare a public key revoked, it remains an open issue how to convey this information to caches that might hold the data. Meanwhile an attacker may continue to replay the old revoked key and signature. The REVKEY and/or revoked bit are only effective if cache learns them.

Because a zone cannot know whether or where its data may have been cached, we believe an effective key resolution

solution should put the responsibility of tracking key status on caching resolvers. The caching resolvers can handle such abrupt key revocation by two techniques: *periodic key re-validation* and *on-demand data re-fetching*. The basic ideas are to let the caches actively synchronize with the zones about their latest public keys, and to use the signature inception time as an implicit sequence number to arbitrate which keys should be used.

1) Periodic key re-validation: For each public key in its cache, a caching resolver can periodically re-fetch it from the corresponding zone's nameservers. If the newly returned key set is different from the cached one and has a later inception time on the signature, the resolver knows that the zone has changed its keys. Given two public key sets that both have valid signatures, the one whose signature has a more recent inception time is more recent, hence should override the other. Once the caching resolver decides to replace the cached public keys with the newly fetched ones, it also deletes all cached data and companion signatures that are signed by the old (revoked) keys.

The signature inception time can also be used to resist replay attacks, in which an adversary replays the revoked key to a caching resolver. Upon detecting a key change, the caching resolver stores the new key's signature inception time until the old key's signature expires. This way, any attempt to replay the old key will be rejected, because either the old key has expired or the cache knows a more recent inception time than the old key. However, we note that key re-validation works only if the attacker cannot intercept and modify on-the-fly traffic between the caching resolver and the zone's authoritative nameserver, or the caching resolver may also consult SecSpider for further validation.

2) On-demand data re-fetching: After a zone changes its keys, a caching resolver will obtain its new keys at the next periodic checking. However, before that, the stub resolvers behind the cache may experience temporary failures in resolving a name in this zone, because the cache does not have the public keys that are needed to verify the cached signatures, as we elaborated in Section IV-A. To address this problem, we enhance the caching resolver with a new capability as follows. When it sends a reply to a stub resolver, it checks whether it has stored the public key that is required to verifying the returned signature. If not, it will query the corresponding zones for the missing public key. If it cannot fetch the specific key from the authoritative nameservers, it deletes the data and signatures in question and re-fetch them from the authoritative nameservers. This way, it can guarantee that whenever it returns a signature, it always has the corresponding public key to verify the signature.

Techniques have been proposed to improve the re-fetching and signal that new data is available. For example, [23] augments the RRSIG inception/expiration values with a numerical *lease* (a number that is based on the zone's SOA serial number) that compliments the lifetime. The lease simply adds a notion that indicates if the zone has not changed its state much since the signature was generated, then resolvers may use the RRSIG's lifetime. If, however, the zone's state (serial number) has transitioned, and exceeded the lease specified by the signature, the record must be flushed from the cache and re-fetched and the resolver should query the zone for any REVKEYs or keys with the revoked bit set.

For example, suppose on April 28^{th} `target.sec`'s SOA serial number were 2008042801 and an operators (*Bob*) signed the zone's records with a expiration date of May 5^{th}, 2008 and a lease of 2009042801. If a day later *Bob* wished to revoke the zone's key, he would simply create a REVKEY and increase the zone's SOA serial to something large, such as 4155526449 (current serial number + 2^{31}). This signals caches that a key revocation may be in progress, and they can then request the REVKEY for `target.sec` and verify this is indeed the case.

E. Handling Dynamic Update

In order to sign the dynamically updated records, some keys must be kept online. However, storing the zone's private key online poses excessive security risks for the zone as well as its descendants. To address this dilemma, we propose a *zone split* technique to minimize potential damages by the compromise of online keys.

Since offline keys provide stronger key protection and online keys are needed for signing dynamic data, we suggest a hybrid approach of utilizing both. Assuming a zone can differentiate the manually updated and the dynamically updated records, it can create a subzone and place all dynamically updated records in this subzone, which we call the *dynamic subzone*. After the zone is split, the private key of the original zone is kept offline, while the private key of the dynamic subzone is kept online at the master server to sign dynamic data.

Note that the hosts do not need to be explicitly re-named to reflect the zone split. Consider the example of the zone *foo.com* and its dynamic subzone *dyn.foo.com*. A host *host.foo.com* with a dynamically updated A record can keep its current name, and add an alias name in the zone: *host.foo.com CNAME host.dyn.foo.com*. The use of such alias names keeps the canonical names stable yet accommodates dynamic updates through one level of indirection. In our example, the name *host.foo.com* is stable, while its associated IP address is changed indirectly through the updates to the A record of *host.dyn.foo.com*. As such, the hosts can keep their old names despite the zone split.

Zone split can achieve the best of both online and offline keys. Since the private key for the dynamic subzone is online, the master nameserver can directly sign the new records after it receives a dynamic update. On the other hand, the security risk of this online key being compromised is minimized. An attacker having access to this online key can forge any records in the dynamic subzone, but he cannot compromise the original zone. This effectively prevents the cascaded damage to the zone's descendants because the delegation records of NS and DS RRs cannot be dynamically updated [31], thus must be placed in the original zone.

VIII. Related Work

Since the seminal work of [6], DNS security has attracted much attention in both research and operation communities. In response to the ever increasing importance of DNS yet its

vulnerabilities to security attacks, IETF has chartered the DNS Extensions (dnsext) Working Group to lead the development of DNSSEC in the past decade. To date, a number of DNSSEC related specifications have been published by IETF. The list includes, but is not limited to, [9], [8], [5], [13]. After years of developments and several rounds of protocol revision, the current set of DNSSEC specifications [2], [4], [3] is believed to be relatively stable and mature, although some aspects of the DNSSEC design continue to evolve [16], [32], [29], [22], [23]. A comprehensive list of DNSSEC documents are maintained on the website http://www.dnssec.net.

However, most of these DNSSEC specifications focus on *what* has been done, rather than *why* we have been doing it. Each of them documents some specific designs for solving individual problem that has manifested as the system evolves. Yet many of the rationales and insights behind these efforts are missing from the public archives, except for those comments loosely documented in the IETF mailing list. In contrast, this paper is the first effort to systematically document the DNSSEC design and deployment issues and classify them in a unified framework. Our study shows that many of these issues are related to each other and can be traced back to a few fundamental properties of the DNS as a large-scale, distributed system. These insights also enable us to propose several practical techniques that can facilitate the DNSSEC rollout and operations.

In a broader context, our case study on DNSSEC reveals that a sound and simple cryptographic design can be very difficult to deploy in a large-scale system. Such a gap between cryptography designs and secure systems has long been recognized [28]. In fact, while we focus on operational issues in this paper, it is shown in [28] that a security system also faces non-trivial challenges in terms of implementation, usability and application integration. However, there is no case study in [28] that illustrates and analyzes these challenges through concrete examples.

Another work directly related to ours is a previous study on the deployment of a large-scale PKI for the United States Department of Defense [20]. Many of the observations in [20] are also applicable in DNSSEC, e.g., the lack of motivation for kicking off the adoption, system maintenance and personnel training. In particular, it shows that key revocation is one of the biggest technical challenges in the PKI deployment and, as a result, the system migrated from the CRL approach to the online query (OCSP) approach. We made similar observation that key revocation is one fundamental challenge in DNSSEC.

IX. Concluding Remarks

It is well known that "security mechanisms are not magic pixie dust that can be sprinkled over completed protocols" [7], and our study serves as a concrete evidence – a simple cryptographic design can face multiple grand challenges when applied to an Internet-scale operational system. While cryptography is widely recognized as a powerful tool for securing the Internet, the exercise of adding cryptographic protection into DNS has proven to be more challenging than anyone had expected. Even in hindsight, the setbacks in the early stages of DNSSEC design are probably inevitable, as DNSSEC is perhaps the first attempt to apply cryptography to an Internet-scale system and the practical challenges in this endeavor were only learned in hard ways. In the process of identifying these challenges, we have also articulated a short list of lessons learned, which we hope would prove useful to future designs of cryptographic protection for other Internet-scale systems.

Lesson 1: *Design for scalability*. One lesson repeatedly observed is that the growth of new technologies or applications tends to be grossly underestimated. A cryptographic design must take great care about operational feasibility as the size of a distributed system grows. The original DNSSEC design's decision of putting the parent's signature of each child zone's public key in the child domain can serve as a thought-provoking lesson. The placement of the signature does not affect the cryptographic correctness, yet it can have profound impact on the required coordination among a large number of administrative domains when a parent zone changes its key.

Lesson 2: *Design for heterogeneity*. The Internet has no centralized authority. As a direct consequence, different administrative domains tend to operate with different practices based on their own judgment regarding the best engineering tradeoff. Cryptographic designers can hardly foresee or arbitrate how the system is implemented or operated. Thus the design should accommodate different practices. For example, the DNSSEC design should not mandate keeping the private keys either online or offline. Instead the design should offer alternatives with operational guidelines to explain the necessary precaution and potential consequence of each practice.

Lesson 3: *Design for incremental deployment*. Incremental deployment is a fundamental requirement for rolling out any new functions in the Internet. Because the Internet is collectively operated by a large number of independent administrative domains, individual parties make their own decisions regarding when to deploy a new function, or whether to deploy it at all, based on their perceived gains and cost. No cryptographic design should rely on universal deployment at once, or in any specific order.

Lesson 4: *Design for imperfect operations*. A large-scale distributed system does not operate in a perfect manner. Rather, inconsistencies, errors and failures exist all the time. A cryptographic design must strive to preserve its protection despite imperfect operations. For example, in an ideal case, a zone's private key should be carefully protected and kept secret. However, it is inevitable that the administrators may make mistakes that may lead to the private key being exposed. Thus, one should design to minimize potential damages of imperfect operations, and design to promptly detect and recover from imperfect operations (e.g., fast key revocation).

One typical place where imperfect operations happen repeatedly is the coordination across administrative domains, thus a cryptographic design should minimize such cross-domain coordination. The evolution of the DNSSEC PKI can serve as a good example. Design changes were made to avoid coordination between a zone and all of its children upon a key change by introducing DS RR, which reduces coordination required between zones. Changes were also made to reduce the frequency of key changes by splitting KSK and ZSK.

Lesson 5: *Design with monitoring as an integral component*. Generally speaking, cryptographic designs for a distributed system require strictly defined operations and coordination among all the components. This directly contradicts the reality of necessarily imperfect operations in an Internet-scale system. Our experience with running SecSpider over the last three years shows that distributed monitoring can be an effective means to observe various operational states and detect possible inconsistencies and errors, thus monitoring should be incorporated as an integral part in the cryptographic design.

REFERENCES

[1] The state and challenges of the dnssec deployment. In *NANOG 44*, 2008. http://nanog.org/meetings/nanog44/presentations/Sunday/Osterweil_DNSSEC_N44.pdf.

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirement. RFC 4033, Mar 2005.

[3] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035, Mar 2005.

[4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034, Mar 2005.

[5] D. Atkins and R. Austein. Threat Analysis of the DNS. RFC 3833, 2004.

[6] S. Bellovin. Using the DNS for System Break-Ins. In *Usenix Security Symposium*, 1995.

[7] S. Bellovin, J. Schiller, and C. Kaufman. Security Mechanisms for the Internet. RFC 3631, 2003.

[8] D. Eastlake. DNS Security Extensions. RFC 2535, 1999.

[9] D. Eastlake and C. Kaufman. DNS Security Extensions. RFC 2065, 1997.

[10] T. I. I. Foundation. .se top level domain. http://www.iis.se/.

[11] IANA. Interim trust-anchor repository. https://itar.iana.org/.

[12] Internet Research Lab, UCLA CS Department. The SecSpider DNSSEC Monitoring Project. http://secspider.cs.ucla.edu/.

[13] O. Kolkman and R. Gieben. DNSSEC Operational Practices. RFC 4641, Sept 2006.

[14] O. Kolkman, J. Schlyter, and E. Lewis. Domain Name System KEY (DNSKEY) Resource Record (RR) Secure Entry Point (SEP) Flag. RFC 3757, 2004.

[15] B. Laurie. Distributing Keys for DNSSEC. Internet Draft, Spet 2004.

[16] B. Laurie, G. Sisson, and R. Arends. DNS Security (DNSSEC) Hash Authenticated Denial of Existence. RFC 5155, Feb 2008.

[17] D. Massey, E. Lewis, O. Gudmundsson, R. Mundy, and A. Mankin. Public Key Validation for the DNS Security Extensions. In *Information Survivability Conference and Exposition (DISCEX II)*, 2001.

[18] P. Mockapetris. Domain Names: Concepts and Facilities. RFC 1034, 1987.

[19] P. Mockapetris. Domain Names: Implementation and Specification. RFC 1035, 1987.

[20] R. Nielsen. Observationsfrom the deployment of a large scale pki. In *4th AnnualPKI R&D Workshop: Multiple Paths to Trust*, 2005.

[21] Nominet. Nominet DNSSEC testbed. http://www.nominet.org.uk/tech/dnssectest/.

[22] E. Osterweil, D. Massey, B. Tsendjav, B. Zhang, and L. Zhang. Security through publicity. In *First USENIX Workshop on Hot Topics in Security*, 2006.

[23] E. Osterweil, V. Pappas, D. Massey, and L. Zhang. Zone state revocation for dnssec. In *ACM Sigcomm Workshop on Large Scale Attack Defenses (LSAD)*, 2007.

[24] E. Osterweil, M. Ryan, D. Massey, and L. Zhang. Quantifying the operational status of the dnssec deployment. In *IMC '08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*. ACM, 2008.

[25] V. Pappas, P. Fltstrm, D. Massey, and L. Zhang. Distributed DNS Troubleshooting. In *ACM SIGCOMM Network Troubleshooting Workshop*, 2004.

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

[27] P. I. R. (PIR). .org top level domain. http://www.iana.org/domains/root/db/org.html.

[28] B. Schneier. Why cryptograph is harder than it looks. White Paper, Counterpane Systems, 1997.

[29] M. StJohns. Signature-Only DNSSEC: A Simplified Approach. Internet Draft, October 2006.

[30] M. StJohns. Automated Updates of DNS Security (DNSSEC) Trust Anchors. RFC 5011, Sept 2007.

[31] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. Dynamic Updates in the DNS. RFC 2136, 1997.

[32] S. Weiler and J. Ihren. Minimally Covering NSEC Records and DNSSEC On-line Signing. RFC 4470, April 2006.

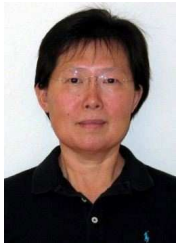
[33] Wikiquote. Yogi berra — wikiquote, 2007. http://en.wikiquote.org/w/index.php?title=Yogi_Berra&oldid=607127.

Hao Yang is a Research Staff Member at IBM T. J. Watson Research Center. He received his Ph.D. in Computer Science from University of California, Los Angeles. His research interests include distributed messaging systems, wireless sensor networking, and network security. He has published over thirty conference and journal papers, including a Best Paper Award from IEEE ICPP 2008.

Eric Osterweil is a PhD candidate at the University of California, Los Angeles. His research focuses on large-scale network measurement systems, network security, and distributed data verification. His thesis work focuses on a concept called the Public-Space.

Dan Massey is an associate professor at Computer Science Department of Colorado State University. Dr. Massey received his doctorate from UCLA and is a senior member of the IEEE, IEEE Communications Society, and IEEE Computer Society. His research interests include protocol design and security for large scale network infrastructures. and he is currently the principal investigator on research projects investigating techniques for improving the Internet's naming and routing infrastructures. He is a co-editor of the DNSSEC standard (RFC 4033, 4034, and 4035).

Songwu Lu is currently an associate professor at UCLA Computer Science. His research interests include wireless networking, mobile systems, sensor networks, network security, and cloud computing.



Lixia Zhang received her Ph.D in computer science from the Massachusetts Institute of Technology and joined the research staff at Xerox Palo Alto Research Center. She joined the faculty of UCLA's Computer Science Department in 1996. In the past she has served as the vice chair of ACM SIGCOMM, and on the editorial board for the IEEE/ACM Transactions on Networking. Zhang is a fellow of ACM and a fellow of IEEE, and the recipient of 2009 IEEE Internet Award.