Quantum-safe Signatureless DNSSEC

Aditya Singh Rawat
Ashoka University
aditya.rawat_phd21@ashoka.edu.in

Mahabir Prasad Jhanwar
Ashoka University
mahavir.jhawar@ashoka.edu.in

Abstract

We present SL-DNSSEC: a backward-compatible protocol that leverages a quantum-safe KEM and a MAC to perform *signature-less* (SL) DNSSEC validations in a single UDP query/response style. Our experiments targeting NIST level I security for QTYPE A query resolution show that SL-DNSSEC is practically equivalent to the presently deployed RSA-2048 in terms of bandwidth usage and resolution speeds. Compared to post-quantum signatures, SL-DNSSEC reduces bandwidth consumption and resolution times by up to 95% and 60%, respectively. Moreover, with *response size* < *query size* $\leq$ 1232 bytes, SL-DNSSEC obviates the long-standing issues of IP fragmentation, TCP re-transmits and DDoS amplification attacks.

1 Introduction

A cryptanalytically relevant quantum computer (CRQC) running Shor's period finding algorithm [64] can efficiently solve the factoring and the discrete logarithm problem (DLP) in polynomial time. Asymmetric schemes, such as RSA and ECDSA, relying on the foregoing hardness assumptions, thus stand in urgent need to be replaced with their quantum-resilient counterparts. While a CRQC can also mount Grover's [33] quadratically faster ($O(\sqrt{2^n})$) brute-force search against symmetric primitives (such as AES and SHA family), the urgency for a post-quantum transition in this case remains less pressing since a doubling of the key length or the hash size restores the original n-bit security.

Many Internet protocols, such as TLS and SSH, rely on public-key cryptography 1) to provide message confidentiality and integrity, and 2) to authenticate the communicating participants. The DNS Security Extensions (DNSSEC) [58–60], being one among such protocols, facilitates the validation (origin authentication and data integrity) of DNS responses with the aid of digital signatures. Being the backbone of the Internet, the Domain Name System (DNS) maps a human-readable domain name (www.example.com) to a machine-understandable IP address (1.2.3.4). At present, DNS services are also utilized for email authentication [39], acquisition of TLS certificates by proving a domain's ownership [10], and supporting Internet routing security (RPKI) [50].

Without DNSSEC in place, DNS remains vulnerable to cache poisoning attacks [1, 11, 12] wherein an adversary can inject a false domain-to-IP mapping in a resolver's cache, thereby eventually redirecting the users of the *poisoned* resolver to a malicious website. In order to perform a successful attack, an off-path adversary would need to simultaneously guess the 16-bit UDP[1] source port and the 16-bit DNS transaction ID. However, recently researchers [51, 52] discovered critical vulnerabilities in DNS software stacks that narrowed this search space from 2^{32} to $2^{16}+2^{16}$, effectively enabling them to compromise resolvers' caches.

[1] DNS primarily uses UDP at the transport layer.

Table 1: A size comparison (in bytes) of signature (sig) / ciphertext (ct) and public key (pk) of various algorithms.

Algorithm	Assumption	Quantum-safe	pk	ct / sig
X25519	ECDLP	✗	32	32
Kyber-512	Lattice	✓	800	768
ECDSA P-256	ECDLP	✗	64	64
RSA-2048	Factoring	✗	260	256
Falcon-512	Lattice	✓	897	666
Dilithium-2	Lattice	✓	1312	2420
SPHINCS$^+$-128s	Hash	✓	32	7856

Although DNS over TLS (DoT) [36], DNS over HTTPS (DoH) [35], and DNS over QUIC (DoQ) [37] have been proposed, it is important to note that they are not a replacement for DNSSEC. The former, being privacy focused, establish an encrypted and authenticated channel between a client and a resolver (*i.e. hop-by-hop* security). On the other hand, DNSSEC 1) operates between resolvers and nameservers, and 2) guarantees the veracity of DNS records by establishing a chain of trust up to the root (*i.e. end-to-end* integrity).

In its endeavour to sustain Internet security in the face of quantum computers, the National Institute of Standards and Technology (NIST) has selected Crystals-Kyber [20] as Key Encapsulation Mechanism (KEM) and Crystals-Dilithium [28], Falcon [56] and SPHINCS$^+$[17] as digital signature algorithms. In comparison to their classical counterparts however, these algorithms (colloquially referred to under the umbrella acronym of PQC — designating Post-Quantum Cryptography), have significantly larger public key and signature / ciphertext sizes as elucidated in Table 1.

Size Constraints on DNS. With the quantum era on the horizon, DNSSEC must transition to CRQC-resistant algorithms so that it may continue to thwart cache poisoning attempts. However, the relatively larger footprint of PQC objects, as discussed above, will have major ramifications on the global DNS infrastructure. A DNS message, as originally specified, was restricted to a size of 512 bytes, with UDP being its primary transport. With a view to DNSSEC's higher space requirements (for transferring signatures and public keys), this size bound was eventually increased to a *theoretical* value of 64 KB with Extension Mechanisms for DNS (EDNS0) [24]. Unfortunately, a DNS packet exceeding the Path Maximum Transmission Unit (PMTU), which is usually 1500 bytes (<< 64 KB), triggers IP fragmentation at the intermediate routers. The ensuing UDP/IP fragments not only may never arrive [19, 68] (*e.g.,* due to being blocked by stateless firewalls) but also can be used to exhaust a resolver's resources [41] or to inject spoofed records in a DNS response [34]. Additionally, the study of [68] has shown that up to 10% of resolvers fail to handle these fragments correctly.

In order to avoid the multitude of issues linked with IP fragmentation, DNS messages are recommended to not exceed **1232** bytes in size [3, 55, 68]. This conservative threshold, derived as 1280 (IPv6

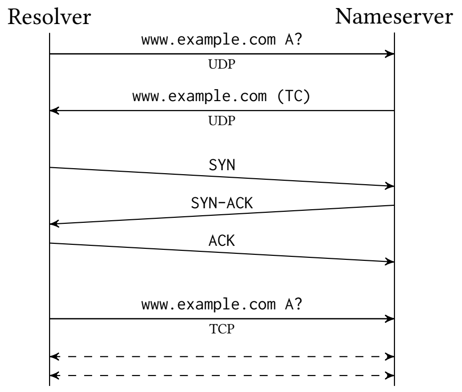

Figure 1: Standard DNS with TCP Fallback

minimum MTU) − 40 (IPv6 Header) − 8 (UDP Header), is deemed to prevent IP fragmentation on almost all network links [9, 54].

For conveying DNS messages that do not fit within the preceding size bracket, the proposed fallback transport is TCP. In Standard DNS flow, a response is marked as truncated (TC-bit is set in the HEADER) if the size thereof exceeds either 1) Resolver's advertised edns-udp-size (*i.e.* the maximum message size it can receive over UDP), or 2) Nameserver's max-udp-size (*i.e.* the maximum message size it can send over UDP). In BIND9 (a popular DNS software), valid values for these parameters range from 512 − 4096. If the PMTU is unknown, a default value of 1232 is used.

A resolver receiving a truncated response, which is a copy of the original query but with TC-bit set, proceeds to discard it (incurring a wasted UDP round-trip) and retries the query (with a new transaction ID) over TCP after performing a three-way handshake with the server. Figure 1 elucidates this flow for a DNSSEC-enabled resolver sending a QTYPE A (IPv4 address) query for www.example.com. The response, which additionally includes one or more PQC signatures, is marked as TC because of exceeding the UDP limits.

Unfortunately, up to 11% of nameservers have been found to lack TCP support by [55, 69]. The report of [55] additionally remarks that TCP/53 connections could even be blocked by intruding middleboxes. In the surveys of [26, 53], a non-trivial number of resolvers did not properly fall back to TCP when requested by nameservers. Lastly, DNS over TCP has been shown to be slower (sometimes by a factor of 4×) and more resource intensive than DNS over UDP [5, 42], thus putting a limit on the number of TCP connections a DNS server might be able to handle concurrently.

Note that a properly implemented TCP support on nameservers and resolvers still does not clear away the road to post-quantum DNSSEC. For *e.g.*, a DNS message containing just three SPHINCS$^+$-256s NIST level V signatures, a common scenario with *non-minimal* QTYPE A responses, even exceeds the maximum possible DNS message size of 64 KB.

1.1 Related Work

Out-of-Band Key Distribution. In [55], Müller *et al.* propose an out-of-band distribution (*i.e.* transportation outside the DNS infrastructure) of large public keys via HTTP or FTP. Unfortunately, not only does this approach require zone-operators to additionally maintain a web server, but it also has been shown to create a resolution overhead of about 30% in [14]. Furthermore, the size complications arising due to PQC signatures still remain unaddressed.

Merkle Tree Ladder (MTL). Fregly *et al.* [30] recently proposed a MTL mode which can reduce the size impact of PQC signatures. Specifically, the signer signs *Merkle tree ladders* that are derived from the messages to be validated. Individual messages are then authenticated relative to the ladder using a Merkle tree authentication path, while the ladder itself is validated using the public key.

Application Layer Fragmentation. With an aim of avoiding the fragility and the unavailability connected with IP fragmentation and TCP fallbacks, respectively, many proposals have been put forward that fragment large DNS messages at the application (DNS) layer. In such a scenario, the nameserver becomes responsible for the fragmentation of a DNS response and the resolver for the subsequent reassembly thereof.

Sivaraman *et al.* [65] fragmented a large DNS response across multiple UDP datagrams, transmitting each fragment sequentially. On the other hand, Additional Truncation Response (ATR) [66] (though not strictly a fragmentation scheme) involved a module which decided whether to send an additional truncated (TC) response (right after the original large response) or not. The basic idea behind ATR was as follows: If the client fails to receive the first large response (for *e.g.*, it gets fragmented at the network layer and the ensuing fragments get dropped by stateless firewalls), the trailing TC response would at least trigger an immediate TCP fallback thereon. Unfortunately, both of these proposals failed to gain traction since multiple responses were being sent out to a single request. Many firewalls are configured with the policy of accepting one response packet per query. Moreover, many resolvers close their sockets immediately after receiving the first response packet. Thus, there were concerns about ICMP flooding since for each trailing response packet that could not be delivered, a *destination unreachable* packet would be sent back to the nameserver.

Addressing the shortcomings of the previous drafts, A Resource Record Fragmentation (ARRF) [32] fragments DNS resource records and sends an additional response only upon an explicit *request*. Since each extra response has its own query, prior concerns about firewalls and ICMP flooding are mitigated. Unfortunately, ARRF fragments, owing to their use of non-standard Type RRFRAG pseudo-records, could be potentially dropped by inspecting middleboxes. Secondly, ARRF remains vulnerable to memory exhaustion attacks, as acknowledged by its authors in [32]. Finally, ARRF requires a minimum of two round-trips to reconstruct the full DNS message.

A recent work, called QNAME-Based Fragmentation (QBF) [57], achieves a one round-trip reassembly of post-quantum DNSSEC messages while using only standard DNS record Type(s). Unlike previous schemes, it fragments *raw* signature and public key bytes stored in RRSIG and DNSKEY records, respectively. The implication is that the fragments *resemble* the original DNS response, except insofar as they carry partial signatures / public keys. A fragment is explicitly *requested* by encoding the desired fragment number in the QNAME field of a query. Lastly, QBF is backwards-compatible and not susceptible to memory-depletion attacks.

Discussion on ARRF / QBF. For fast query resolutions, both ARRF (in 2nd round trip) and QBF (in 1st round trip) send multiple DNS over UDP messages in parallel. On busy resolvers and nameservers, handling thousands of queries per second, this deluge of DNS packets could lead to a starvation of network bandwidth. Moreover, such bursts in traffic can conceivably overwhelm load balancers or trigger flood protection in firewalls. This is because unlike its TCP sibling, UDP does not have any built-in flow and congestion control mechanisms.

Furthermore, it is crucial to remember that UDP/IP does not guarantee a reliable delivery of packets. In ARRF/QBF, as the number of signatures to transmit or the sizes thereof grow (from setting higher NIST levels), the number of DNS messages that need to be exchanged also inevitably rises. Therefore, the probability of at least one DNS query/response packet getting dropped during transit also increases, resulting in unforeseen resolution delays or timeouts.

To give a perspective, considering a 1% network loss rate and SPHINCS$^+$-128s as the zone signing algorithm, the probability of at least one ARRF/QBF packet being lost during transit can be calculated as $\Pr = 1 - (0.99)^{46} = 0.37$, where 46 is the (approximate) total number of DNS packets exchanged during the session. This implies that, with a one-third probability, a ARRF/QBF SPHINCS$^+$ session will require an extra round-trip. While the picture is not as bleak with Falcon and Dilithium, it is circumspect to be prepared for all circumstances, especially since SPHINCS$^+$ still remains the most conservative choice among its siblings.

Another concern with ARRF/QBF is their potential to be exploited for a DDoS attack [40, 61, 70], wherein small DNS over UDP queries with a spoofed source IP address cause large DNS responses (amplification) to be sent out from a server to a target IP device (reflection), eventually overwhelming the latter or the network thereof. In one of the major DDoS events, the attackers were able to generate 300 Gbps of traffic on a Tier 1 provider using open DNS resolvers [2]. On a related note, performing such type of attacks over TCP is not feasible because of the three-way TCP handshake. This is because client's query is forwarded to the DNS software only after receiving a valid[2] client ACK to the server SYN.

Bearing the above apprehensions in mind, it appears that fragmentation schemes may not be the panacea for DNSSEC's complications in the quantum age. Therefore, in this work, we take a fundamentally different approach by performing DNSSEC validations without PQC signatures. More precisely, we leverage the concept of authentication via a key exchange.

Authenticated Key Exchange without Signatures. The notion of an authenticated key exchange (AKE) follows a long succession of works, with the early proposals being [16, 21]. In the SKEME protocol [45] and the RSA key-transport (in TLS versions up to 1.2), an entity is authenticated via a successful decryption of a challenge message. The protocol of Bellare *et al.* [15] obtained authentication from long-term Diffie-Hellman (DH) keys. In particular, the resulting shared secret is fed into the session key calculation to derive an *implicitly* authenticated key (*i.e.* only the legitimate parties could compute it). Other DH-based AKE protocols include MQV [49], HMQV [46], NAXOS [48], Noise [6], Signal [4] and WireGuard [27]. Constructions using generic Key Encapsulation Mechanisms (KEMs) for AKE have also been proposed in [25, 31].

In the domain of TLS 1.3, the OPTLS proposal [47] is a DH-based AKE that offers a signature-free handshake. Specifically, the server sends a certificate containing a DH public key whilst combining the corresponding long-term secret key with the ephemeral public key from the client. The resulting shared key is then used to generate a MAC which authenticates the server. Unfortunately, there does not yet exist an efficient OPTLS instantiation for a post-quantum setting ([72], Ch. 12). KEMTLS [62], which builds upon OPTLS, is a KEM-based AKE that bypasses the usual signed-DH flow of TLS to achieve a signature-less PQC handshake. More concretely, the client performs an encapsulation against the server's KEM public key (obtained via the `ServerCertificate` message during the handshake) to derive an *implicitly* authenticated shared secret, which is then used to encrypt the first flight of application data from the client. The server is later *explicitly* authenticated with the `ServerFinished` message. Note that to validate the server's KEM public key, the client still unavoidably relies upon a CA signature. A follow-up work by the same authors, called KEMTLS-PDK [63], is a variant of KEMTLS that uses pre-distributed keys for earlier authentication. This scenario occurs when a web-browser caches certificates or in the case of Internet of Things (IoT) devices or mobile applications that come with pre-bundled certificates.

1.2 Our Contributions

Given the practical size constraint on DNSSEC messages that impedes a smooth adoption of post-quantum cryptography, we show how an authenticated key exchange (AKE) can be used to achieve a signature-free validation of DNS resource records. To this end, we propose SL-DNSSEC: an AKE-based protocol for DNSSEC which uses 1) A quantum-safe KEM to first establish a shared key between a resolver and a nameserver, and 2) a Message Authentication Code (MAC), computed under the shared key, to simultaneously authenticate a DNS record's origin and verify its integrity.

An overview of the protocol is illustrated in Figure 2. The resolver holds a DNSSEC-validated KEM public key of the nameserver. Using a series of KEM and Key Derivation Function (KDF) operations, both parties derive a symmetric MAC key. The nameserver sends a MAC tag instead of a signature on the answer record.

We now outline the salient benefits of SL-DNSSEC, with a summary thereof in Table 2. All numerical values below have been inferred from Tables (11, 12) in §5.2.1 of this paper.

- **Massive bandwidth savings.** Compared to SPHINCS$^+$, Dilithium and Falcon, SL-DNSSEC transfers about 95%, 86%, and 58% less data (Fig. 3) during a QTYPE A query lookup.
- **Fast 1-RTT resolution.** SL-DNSSEC remains 50% – 60% faster than DNSSEC over Standard DNS (SD), with the latter incurring the speed penalty of a wasted UDP round-trip and then of a three-way TCP handshake.
- **One packet sent/received.** Although ARRF/QBF take two and one round-trip(s), respectively, they exchange multiple packets (up to 46) in parallel, thereby increasing the chances of packet drops and UDP flooding. SL-DNSSEC, however, sends only a single query/response.

[2] With `Acknowledgment Number` = `Server Sequence Number + 1`

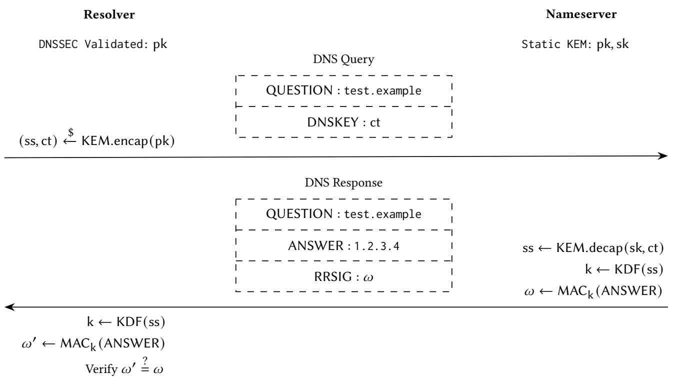

Figure 2: An abstracted view of SL-DNSSEC validating the answer IP 1.2.3.4. The resolver has already fetched and DNSSEC-validated the KEM public key of the nameserver.

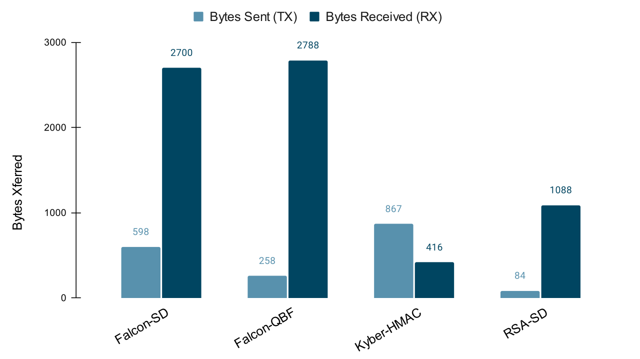

Figure 4: A transmit (TX) / receive (RX) bandwidth usage comparison between Kyber-HMAC (SL-DNSSEC) and signature-based DNSSEC methods. SD denotes Standard DNS.

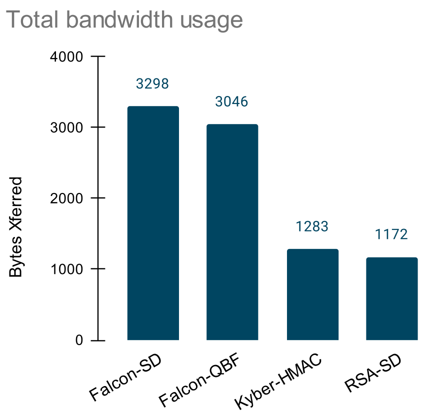

Figure 3: A total bandwidth usage comparison between Kyber-HMAC (SL-DNSSEC) and signature-based DNSSEC methods. SD denotes Standard DNS.

- **DDoS amplification/reflection mitigation.** With *smaller* responses than queries (Fig. 4), SL-DNSSEC fulfils the take-away 6 in the vision paper [40] of being an *“amplification-resistant solution for post-quantum DNSSEC"*.
- **Backward compatibility.** SL-DNSSEC uses standard record Type(s) and wire format to ensure that messages pass through stringent firewalls. It also allows for a graceful fallback to regular DNSSEC flow should one of the endpoints be protocol-oblivious. Moreover, a zone can deploy SL-DNSSEC without needing its parent to be protocol-aware.

To evaluate SL-DNSSEC, we program a daemon that can run atop any DNS provider (such as BIND9, PowerDNS, etc.). The daemon performs all SL-DNSSEC-related operations on behalf of the DNS software. In fact, no changes to the underlying DNS stack are required, except for a small patch on the resolver’s side to detect the Z-bit in the DNS HEADER.

Table 2: A comparison between SL-DNSSEC and signature-based DNSSEC methods. SD : Standard DNS (TCP Fallback).

	SL-DNSSEC	DNSSEC over SD	DNSSEC over ARRF/QBF
No TCP fallback	✓	✗	✓
Low bandwidth usage	✓	✗	✗
Fast resolution	✓	✗	✓
DDoS amp. resistant	✓	✓	✗
No network flooding	✓	✓	✗
1 packet sent/recvd.	✓	✗	✗
Reliability	✓	✓	✗

2 Preliminaries

Notations. The term *resource record* (RR) is often referred to as simply a *record*. || represents concatenation. X ⟶ Y denotes member Y of an abstract structure X. In the context of networking protocols, A/B indicates A over B (*e.g.,* DNS/UDP — DNS (Application layer) over UDP (Transport layer)). RTT stands for round-trip time. ANS is short for Authoritative Name Server. For presentation, we omit the root label (*i.e.* the trailing period (.) as in `example.com.`) while writing fully qualified domain names (FQDNs). The word *transfer* is occasionally abbreviated as *xfer*. The Bandwidth Amplification Factor (BAF) of a DNS over UDP session is calculated as:

$$\text{BAF} = \frac{\text{Number of bytes received (RX)}}{\text{Number of bytes sent (TX)}}$$

2.1 Domain Name System (DNS)

We briefly review the relevant background on DNS. Consider a canonical domain name: `www.example.com.` (with the trailing dot). Each label: (`www`), (`example`), (`com`) and (`.`)[3] corresponds to a level within the DNS hierarchy, with the root (`.`) being at the apex. Under the root come *top-level domains* or TLDs (`com`), and within these are *second-level* domains (`example`), and then *subdomains* (`www`).

[3] The root label is technically `null`.

Table 3: DNS HEADER **Wire Format**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ID															
QR	OpCode				AA	TC	RD	RA	Z	AD	CD	RCode			
QDCount															
ANCount															
NSCount															
ARCount															

- ID: used by requester to match a response to its query
- QR: whether message is a query (0) or a response (1)
- AA: whether response is authoritative (1) or not (0)
- TC: whether response is truncated (1) or not (0)
- Z: a reserved bit set to 0 by default
- AD: whether response has authenticated data (1) or not (0)
- RCode: (0) or NOERROR; (1) or FORMERR – malformed query; (2) or SERVFAIL – server failure; (3) or NXDOMAIN – domain name does not exist

A nameserver that contains definitive information for the zone is said to be *authoritative* for the zone. For *e.g.,* `example.com` ANS is authoritative over the A record for `www.example.com`.

DNS Lookup. To retrieve the IP address of `www.example.com`, the client (*stub resolver*) sends a *recursive* QTYPE A DNS query to its resolver (local DNS server). The resolver, in the event of not having the answer in its cache, performs the following steps *iteratively*:

(1) It sends a QTYPE NS query to a root `(.)` nameserver, which subsequently responds with the following *glue* (referral) records: 1) A Type NS record containing the domain name of com nameserver 2) A Type A record containing the IP address of com nameserver.
(2) It sends a QTYPE NS query to the com nameserver, which subsequently responds with the following *glue* records: 1) A Type NS record containing the domain name of `example.com` nameserver 2) A Type A record containing the IP address of `example.com` nameserver.
(3) It sends a QTYPE A query to the `example.com` nameserver, which finally responds with a Type A record containing the IP address of `www.example.com`.
(4) It caches and forwards the received IP to the client.

DNS Wire Format. A generic DNS message is divided into 5 sections: HEADER, Question, Answer, Authority, and Additional. The HEADER is always present and has a constant size of 12 bytes. Table 3 presents the wire format of a DNS HEADER. The Question section consists of the following fields: QNAME (specifies the domain name encoded in the standard DNS name notation. For *e.g.,* `test.example` is encoded as `[4]test[7]example[0]`), QTYPE (specifies the type of DNS records being requested), and QCLASS (specifies the class of the query, by default set to IN *i.e.* Internet). The last three sections (Answer, Authority, and Additional) have the same format: a possibly empty list of concatenated DNS records.

The DNS resource records (RRs) are database entries that provide information about a domain name. Each record has the following sections: NAME (specifies the domain name encoded in standard notation), TYPE (indicates the type of RR), CLASS (specifies the class of data, defaults to IN), TTL (time-to-live in seconds *i.e.* how long the RR can stay cached), RDLENGTH (specifies the length in bytes of the RDATA field), and RDATA (contains the actual data associated with the record). The Type A and AAAA records contain IPv4 and IPv6 addresses in their RDATA fields, respectively. The Answer section contains records that answer the question; the Authority section contains records that point toward an ANS; the Additional section contains records which relate to the query, but are not strictly answers to the question.

OPT Record. EDNS0 [24] introduces a pseudo-record called OPT (short for *options*) in the Additional section of a DNS message. Note that unlike traditional resource records, pseudo-records do not actually exist in a zone file and are instead created on-the-fly. In queries, a requester specifies the maximum DNS message size it is willing to accept (also known as EDNS0 buffer size or UDP payload size) in OPT → CLASS. In addition to this, the requester also indicates its ability to handle DNSSEC records by setting the DO (DNSSEC OK) bit in OPT → TTL.

OPT → RDATA contains DNS cookies [29] which provide limited security against common off-path attacks such as denial-of-service (server resource exhaustion, amplification/reflection, etc.), cache poisoning and answer forgery. Fundamentally, the cookies serve to 1) add entropy to DNS messages, and 2) verify the IP ownership of the client. The cookies are computed as:

- Client Cookie (8 bytes) =
  Hash(Client IP || Server IP || Client Secret)
- Server Cookie (8-32 bytes) =
  Hash(Client IP || Client Cookie || Server Secret)

2.2 DNS Security Extensions (DNSSEC)

DNSSEC enhances the security of DNS by ensuring the authenticity and integrity of resource records. To realize this aim, it introduces three[4] new types of resource records: Resource Record Signature (RRSIG), DNS Public Key (DNSKEY), and Delegation Signer (DS).

1) RRSIG. A digital signature is computed, using a secret key (discussed below) over a set (called an RRset) of DNS resource records that have the same NAME, CLASS and TYPE. The resulting signature is stored in the RDATA → `Signature` field of an RRSIG record (refer Table 4).

2) DNSKEY. A DNSKEY record (Table 5) stores a public key. Each zone employs two types of keys: Zone Signing Key (ZSK) and Key Signing Key (KSK). KSK is used to sign only DNSKEY RRsets while ZSK is used to sign all other RRsets. Whenever a resolver receives a DNS response with an RRSIG, it uses the associated DNSKEY record to verify the signature contained therein.

3) Delegation Signer (DS). The DS record (Table 6) plays a pivotal role in *recursively* constructing a secure chain of trust from a child zone to the DNS root (.).

When a resolver verifies RRSIGs using the ZSK_{pk} of a child zone, it must also ascertain the authenticity of that key. Recall that the DNSKEY RRset containing ZSK_{pk} and KSK_{pk} is signed using the child's KSK_{sk}. Since KSK is ultimately self-signed, a resolver must

[4] A fourth Type NSEC(3) record, used to verify the non-existence of a record name and type, is outside the purview of this work.

Table 4: RRSIG **Wire Format**

RRSIG Record				
NAME	TYPE = RRSIG	CLASS	TTL	RDLENGTH
RDATA				
Type Covered	Type of records signed			
Algorithm	Signature algorithm used			
Labels	Number of labels in the signed name			
Original TTL	Original time-to-live of the records signed			
Signature Expiration	When the signature expires			
Signature Inception	When the records were signed			
Key Tag	Key ID to be used for signature verification			
Signer's Name	Name of the signer			
Signature	← sign(RRSIG → RDATA‖RR(1)‖RR(2)‖ . . .) where RDATA excludes Signature and RR(i) is the i-th record in the RRset			

Table 5: DNSKEY **Wire Format**

DNSKEY Record				
NAME	TYPE = DNSKEY	CLASS	TTL	RDLENGTH
RDATA				
Flags	Specifies whether the key is a ZSK (256) or a KSK (257)			
Protocol	Always set to `0x03` to indicate DNSSEC			
Algorithm	Signature algorithm of the key			
Public Key	Contains the raw public key bytes			

Table 6: DS **Wire Format**

DS Record				
NAME	TYPE = DS	CLASS	TTL	RDLENGTH
RDATA				
Key Tag	ID of the KSK which is hashed			
Algorithm	Signature algorithm of the key			
Digest Type	Hash algorithm			
Digest	← hash(DNSKEY → NAME ‖ DNSKEY → RDATA)			

also connect the trust thereof with the child's parent. To specifically aid resolvers in this endeavour, the child generates a cryptographic hash of its $\mathsf{KSK_{pk}}$ and shares it with its parent in a DS record.

During a DNS lookup, when a resolver is referred to a child zone by its parent, the latter provides a DS record containing the hash of the child's $\mathsf{KSK_{pk}}$. This DS record is what indicates to the resolver that the child zone is DNSSEC-enabled. Importantly, the parent also furnishes an RRSIG on this DS record using its own $\mathsf{ZSK_{sk}}$.

To validate the child zone's $\mathsf{KSK_{pk}}$, the resolver hashes it and compares it to the DS record from the parent. Additionally, the resolver also verifies the associated RRSIG of that DS record using the $\mathsf{ZSK_{pk}}$ of the parent.

DNSSEC Lookup. This is similar to the DNS lookup process described in §2.1, except that the resolver now sets the DO (DNSSEC OK) bit in its DNS query. The following extra records are therefore returned at each step:

(1) The root (.) nameserver also sends `com`'s DS and RRSIG thereon created with (.)'s $\mathsf{ZSK_{sk}}$. Additionally, it sends (on an explicit QTYPE DNSKEY query) (.)'s DNSKEYs and RRSIG thereon created with (.)'s $\mathsf{KSK_{sk}}$. Here, we assume the resolver already holds (.)'s $\mathsf{KSK_{pk}}$ as the *trust anchor*.

(2) The `com` nameserver also sends `example.com`'s DS and RRSIG thereon created with `com`'s $\mathsf{ZSK_{sk}}$. Additionally, it sends (on an explicit QTYPE DNSKEY query) `com`'s DNSKEYs and RRSIG thereon created with `com`'s $\mathsf{KSK_{sk}}$.

(3) The `example.com` nameserver also sends RRSIG created with its $\mathsf{ZSK_{sk}}$ on the Type A record containing the answer IP. Moreover, it sends (on an explicit QTYPE DNSKEY query) its DNSKEYs and RRSIG thereon created with its $\mathsf{KSK_{sk}}$.

On a successful DNSSEC validation, the resolver sends its answer response to the client with HEADER → AD set.

2.3 Key Encapsulation Mechanism (KEM)

Definition 1. *A Key Encapsulation Mechanism (KEM) is an asymmetric primitive that allows two parties to establish a shared secret in a key space* $\mathcal{K}$.

A KEM instance defines three probabilistic operations:

- **Key Generation**: KEM.keygen() generates a public and private keypair (pk, sk).
- **Encapsulation**: KEM.encap(pk) generates a shared secret ss in a key space $\mathcal{K}$ and a ciphertext (encapsulation) ct against pk.
- **Decapsulation**: KEM.decap(sk, ct) takes as input sk and ct, and decapsulates the shared secret $\mathsf{ss}' \in \mathcal{K}$. In a δ-correct scheme, $\Pr(\mathsf{ss} = \mathsf{ss}') \geq 1 - \delta$.

Security Model. Shared secret (ss) should be indistinguishable from random (IND), given just pk (Chosen Plaintext Attack (CPA)) or additionally given access to a decapsulation oracle (Chosen Ciphertext Attack (CCA)).

3 The SL-DNSSEC Protocol

SL-DNSSEC is a backward-compatible and amplification-resistant protocol for DNSSEC that validates DNS resource records without signatures. Pursuant to this objective, it uses as its primary building blocks: 1) A post-quantum KEM to first establish a shared secret between a resolver and a nameserver 2) A KDF to derive a symmetric MAC key of an appropriate length from the shared secret, and 3) A MAC to compute authentication tags on DNS records.

On a high-level, a nameserver generates a KEM keypair and adds the public key to its DNSKEY RRset, re-signing the latter with its $\mathsf{KSK_{sk}}$. This RRset is fetched and DNSSEC-validated by a resolver using the covering RRSIG and a signed DS record from the zone's parent. For any subsequent interaction, the resolver and the nameserver compute a symmetric MAC key using KEM and KDF operations. The nameserver then sends MAC tags instead of signatures on DNS RRsets.

We now demonstrate the execution of the SL-DNSSEC protocol between a resolver and an ANS. Note that SL-DNSSEC can also be deployed on other zones, such as (`.`) or `com`.

The protocol is broadly divided into four phases.

3.1 Phase 1: KEM Key Generation

Assume a DNSSEC-enabled zone (say, `example.com`) with a Key Signing Key ($\mathsf{KSK_{pk}}$, $\mathsf{KSK_{sk}}$) and a Zone Signing Key ($\mathsf{ZSK_{pk}}$, $\mathsf{ZSK_{sk}}$). Therefore, the current public key RRset of `example.com` comprises: 2 Type DNSKEY records containing $\mathsf{KSK_{pk}}$ and $\mathsf{ZSK_{pk}}$, respectively.

Table 7: An abstracted view of a QTYPE `DNSKEY` response containing the public keys of `example.com` zone

Header Section
Question Section
QNAME = `example.com`
QTYPE = `DNSKEY`
QCLASS = `IN`
Answer Section
DNSKEY $\mathsf{KSK_{pk}}$
DNSKEY $\mathsf{ZSK_{pk}}$
DNSKEY $\mathsf{ZKK_{pk}}$
RRSIG with $\mathsf{KSK_{sk}}$
Authority Section
Additional Section
OPT

The zone operator now runs KEM.keygen() to generate a *Zone KEM Key*: $(\mathsf{ZKK_{pk}}, \mathsf{ZKK_{sk}})$.

Thereafter, the operator performs the following steps:

(1) Create a generic DNSKEY record.
(2) Set DNSKEY $\longrightarrow$ `RDATA` $\longrightarrow$ `Flags` = 258
(3) Set DNSKEY $\longrightarrow$ `RDATA` $\longrightarrow$ `Algorithm` = KEM
(4) Set DNSKEY $\longrightarrow$ `RDATA` $\longrightarrow$ `Public Key` = $\mathsf{ZKK_{pk}}$
(5) Add DNSKEY to the existing RRset of public keys.
(6) Re-sign the RRset using $\mathsf{KSK_{sk}}$.

Here, the value 258 for `Flags` is one of the available choices after turning off the Secure Entry Point (SEP) bit (refer RFC [60] §2.1.1). Note that the SEP flag is set only for a KSK which has a DS record in the parent zone. Furthermore, in some DNS software, the signature over a public key RRset is computed using both $\mathsf{KSK_{sk}}$ and $\mathsf{ZSK_{sk}}$, thus resulting in two RRSIGs (consult [23], §4.7).

When a DNS resolver now sends a QTYPE `DNSKEY` query to `example.com`, it will receive a DNS response (consult Table 7 for its structure) containing the following records:

– Three DNSKEY records holding $\mathsf{KSK_{pk}}$, $\mathsf{ZSK_{pk}}$ and $\mathsf{ZKK_{pk}}$, together constituting 1 RRset
– One covering RRSIG using $\mathsf{KSK_{sk}}$ on the RRset

The implication of the RRSIG is that the trust of the KEM key (ZKK) can now be established with the zone's parent and then recursively with the root. This is easy to see since the `com` zone (the parent of `example.com`) already holds a DS record bearing the hash of `example.com`'s $\mathsf{KSK_{pk}}$.

The resolver thereupon verifies the RRSIG using $\mathsf{KSK_{pk}}$ and then validates $\mathsf{KSK_{pk}}$ itself via the signed DS record it had earlier received from the `com` nameserver during the referral.

Finally, observe that the `com` zone and the root are not required to be SL-DNSSEC-aware during the entire phase.

3.2 Phase 2: Preparing a SL-DNSSEC Query

Assume that a resolver intends to send a QTYPE `A` query with a QNAME `www.example.com` to the `example.com` ANS. We additionally presume that the resolver has already fetched and validated the QTYPE `DNSKEY` response (as outlined in §3.1) from the ANS.

The aforesaid is a common scenario in DNS, wherein a resolver already stores the DNSKEYs of previously contacted zones in its cache. Alternatively, the resolvers can retrieve the public keys of an unacquainted zone first, before dispatching their main query.

Table 8: Wire format: SL-DNSSEC **Query** Q **containing the KEM ciphertext** ct

Header Section
Question Section
QNAME = `www.example.com`
QTYPE = `A`
QCLASS = `IN`
Answer Section
Authority Section
Additional Section
NAME = `example.com` TYPE = `DNSKEY` ⋮ RDLENGTH = x RDATA │ Flags = $\mathsf{ZKK_{ID}}$ │ Protocol = `0x03` │ Algorithm = `KEM` │ Public Key = `0x2a4b`…(ct)
OPT

The resolver now executes the following operations:

(1) Create a QTYPE `A` query message, say Q.
(2) Perform a KEM encapsulation against $\mathsf{ZKK_{pk}}$ of `example.com` to probabilistically get a shared secret ss and ciphertext ct.

$$(\mathsf{ss}, \mathsf{ct}) \overset{\$}{\longleftarrow} \mathsf{KEM.encap}(\mathsf{ZKK_{pk}})$$

(3) Create a generic DNSKEY record.
(4) Set DNSKEY $\longrightarrow$ `RDATA` $\longrightarrow$ `Flags` = $\mathsf{ZKK_{ID}}$
(5) Set DNSKEY $\longrightarrow$ `RDATA` $\longrightarrow$ `Algorithm` = KEM
(6) Set DNSKEY $\longrightarrow$ `RDATA` $\longrightarrow$ `Public Key` = ct
(7) Insert DNSKEY in Q $\longrightarrow$ Additional section.
(8) Send Q.

In *rare* settings wherein a zone offers multiple KEM public keys, a resolver additionally needs to include information about the particular KEM key it has used for encapsulation. To this end, we repurpose the 2-byte `Flags` field. Specifically, the resolver computes the 2-byte `Key Tag`, say $\mathsf{ZKK_{ID}}$, using the Type `DNSKEY` record of the utilized KEM key (refer RFC [60], Appendix B for the algorithm used for `Key Tag` computation) and sets `Flags` = $\mathsf{ZKK_{ID}}$ as previously outlined.

An example wire format of the resulting DNS query Q is illustrated in Table 8. Notice that Q transports ciphertext ct using the standard DNSKEY[5] record and wire format. Moreover, inserting DNSKEY record (bearing ct) in the Additional section further improves backward compatibility, since a SL-DNSSEC-oblivious ANS would ignore it and proceed with the usual DNSSEC flow.

3.3 Phase 3: Preparing a SL-DNSSEC Response

On receiving the DNS query Q, the `example.com` ANS executes the following actions:

[5] The `Public Key` field is opaque (*i.e.* its content is not meaningful) to middleboxes. Semantically, ct is an encapsulated shared secret *key*.

Table 9: An abstracted view of a non-minimal DNS response to a QTYPE A query

Header Section
Question Section
QNAME = `www.example.com`
QTYPE = `A`
QCLASS = `IN`
Answer Section
RR_1 TYPE `A`
$RRSIG_1$
Authority Section
RR_2 TYPE `NS`
$RRSIG_2$
Additional Section
RR_3 TYPE `A`
$RRSIG_3$
OPT

(1) Prepare a traditional QTYPE `A` DNS response (say, R) to Q. In this example, we assume R to be a *non-minimal* DNS response (refer Table 9 for its general format) which contains the following records[6]:
  (a) 1 Type `A` resource record (RR_1) in Answer section containing the answer IP address and 1 covering $RRSIG_1$
  (b) 1 Type `NS` record (RR_2) in Authoritative section containing the nameserver's name and 1 covering $RRSIG_2$
  (c) 1 Type `A` record (RR_3) in Additional section containing the nameserver's IP address and 1 covering $RRSIG_3$

(2) Check if the size of R is within:
  (a) Resolver's UDP *receive* limit, as publicized in Q $\longrightarrow$ OPT $\longrightarrow$ `CLASS`
  (b) Nameserver's UDP *send* limit, as configured in `named.conf`

  – If affirmative, the ANS has the option to send R *as it is* (*i.e.* with signatures). In this example, we presume this check to return negative (which is expected with PQC signatures).

(3) Check for a DNSKEY record containing a KEM ciphertext in Q $\longrightarrow$ Additional section.
  – If negative, continue with the regular DNSSEC flow. Otherwise, proceed as below.

(4) Extract the ciphertext ct from DNSKEY record and do a KEM decapsulation using ZKK_{sk} to obtain the shared secret ss.

$$\mathsf{ss} \longleftarrow \mathsf{KEM.decap}(\mathsf{ZKK_{sk}}, \mathsf{ct})$$

  – If the ANS holds multiple KEM ZKKs, the correct key for the decapsulation can be identified using the `Key Tag` (ZKK_{ID}) provided by the resolver in the `Flags` field (see §3.2).

(5) Feed ss to a secure KDF to derive a key k of requisite length.

$$\mathsf{k} \longleftarrow \mathsf{KDF}(\mathsf{ss})$$

(6) For every $RRSIG_i$ in response R, do:
  (a) Set $RRSIG_i \longrightarrow$ `RDATA` $\longrightarrow$ `Algorithm` = `KEM`
  (b) Set $RRSIG_i \longrightarrow$ `RDATA` $\longrightarrow$ `Key Tag` = ZKK_{ID}
  (c) Let msg := $RRSIG_i \longrightarrow \mathsf{RDATA} \| RR_i(1) \| RR_i(2) \| \ldots$

Table 10: Wire format: Original response with signatures (Left), SL-DNSSEC response with MACs (Right)

Header Section	**Header Section**
Question Section	**Question Section**
QNAME = `www.example.com`	QNAME = `www.example.com`
QTYPE = `A`	QTYPE = `A`
QCLASS = `IN`	QCLASS = `IN`
Answer Section	**Answer Section**
NAME = `www.example.com`	NAME = `www.example.com`
TYPE = `A`	TYPE = `A`
⋮	⋮
RDLENGTH = 4	RDLENGTH = 4
RDATA = `1.2.3.4`	RDATA = `1.2.3.4`
NAME = `www.example.com`	NAME = `www.example.com`
TYPE = `RRSIG`	TYPE = `RRSIG`
⋮	⋮
RDLENGTH = x	RDLENGTH = y
RDATA	RDATA
Type Covered = `A`	Type Covered = `A`
Algorithm = `FALCON`	Algorithm = `KEM`
⋮	⋮
Key Tag = ZSK_{ID}	Key Tag = ZKK_{ID}
Signer's Name = `example.com`	Signer's Name = `example.com`
Signature = `0x1a2b`…(σ_1)	Signature = `0xfae5`…(ω_1)
Authority Section	**Authority Section**
⋮	⋮
⋮	⋮
Signature = `0x3c4d`…(σ_2)	Signature = `0xd4cf`…(ω_2)
Additional Section	**Additional Section**
⋮	⋮
⋮	⋮
Signature = `0x5e6f`…(σ_3)	Signature = `0xb2ac`…(ω_3)
OPT	OPT

  – where `RDATA` excludes `Signature` and $RR_i(j)$ is the j-th resource record in $RRset_i$
  (d) Compute $\omega_i \longleftarrow \mathsf{MAC_k}(\mathtt{msg})$
  (e) Set $RRSIG_i \longrightarrow$ `RDATA` $\longrightarrow$ `Signature` = ω_i

(7) Send R.

Table 10 depicts a comparison between the original DNS response containing PQC signatures (here, Falcon) and the SL-DNSSEC response containing MACs. Note that `RDLENGTH` y << `RDLENGTH` x since MACs are usually much *smaller* than post-quantum signatures. The implication here is that as the number of RRSIGs increase, the size disparity between a signature-based response and its SL-DNSSEC counterpart becomes even more exaggerated.

A noteworthy distinction between SL-DNSSEC and the conventional DNSSEC flow is also herein encountered. While in the latter approach, signatures are usually pre-generated (*i.e.* the zone file is signed offline and then published on the nameserver), the former computes MACs *on-the-fly*[7].

Finally, observe that a MAC is computed over exactly the same message as that specified for a signature in the DNSSEC RFC (refer

[6] For simplicity, here each RRset contains only 1 resource record.

[7] Analogous to Cloudflare's DNSSEC live signing with ECDSA P-256.

§3.1.8.1. in [60]). Additionally, the response R uses the standard RRSIG[8] record and wire format.

3.4 Phase 4: Validating a SL-DNSSEC Response

In due course, when the resolver receives the DNS response R, it proceeds to validate the resource records contained therein in the following manner:

(1) Fetch the shared secret ss from the state.
(2) Feed ss to the KDF to derive the key k.

$$\mathsf{k} \longleftarrow \mathsf{KDF}(\mathsf{ss})$$

(3) For every RRSIG_i in response R, do:
   (a) Check $\mathsf{RRSIG}_i \rightarrow \mathtt{RDATA} \rightarrow \mathtt{Algorithm}$
   — If a signature algorithm is detected, execute the usual signature validation flow. If a KEM algorithm is found, proceed as below.
   (b) Let $\mathtt{msg} := \mathsf{RRSIG}_i \rightarrow \mathtt{RDATA} \| \mathsf{RR}_i(1) \| \mathsf{RR}_i(2) \| \ldots$
   — where RDATA excludes Signature and $\mathsf{RR}_i(j)$ is the j-th resource record in RRset_i
   (c) Compute $\omega'_i \longleftarrow \mathsf{MAC}_\mathsf{k}(\mathtt{msg})$
   (d) Verify $\omega'_i \stackrel{?}{=} \mathsf{RRSIG}_i \rightarrow \mathtt{RDATA} \rightarrow \mathtt{Signature}$
(4) If all RRSIGs are verified, mark R as secure.

3.5 Backward Compatibility

We now examine what happens when only one of the end points implements the SL-DNSSEC protocol while the other one does not.

- *Protocol-aware Requester | Protocol-oblivious Responder:* The requester will not find a KEM $\mathsf{ZKK_{pk}}$ in the QTYPE DNSKEY response from the server. It will then send a plain DNS query.

- *Protocol-oblivious Requester | Protocol-aware Responder:* A KEM $\mathsf{ZKK_{pk}}$ (along with $\mathsf{KSK_{pk}}$ and $\mathsf{ZSK_{pk}}$) would be sent to the requester in the QTYPE DNSKEY response. However, $\mathsf{ZKK_{pk}}$ would be ignored as a key with an *unsupported* algorithm. The requester will then dispatch a usual DNS query. The responder, on not finding a KEM ciphertext in the query, will then proceed with the regular DNSSEC flow. In due time, when the requester receives a DNS response containing signatures, it will pick the relevant key (*i.e.* $\mathsf{ZSK_{pk}}$) to perform the validation of resource records.

4 Security

We assess SL-DNSSEC's security under the standard attacker model as used in a previous DNSSEC study [13]. In particular, the adversary's ultimate aim is to induce the resolver to accept a malicious answer in Phase 4 (§3.4) of the protocol. All the capabilities of the (on-path) adversary, or lack thereof, are as listed below:

— It may eavesdrop on any exchanged packet.
— It may intercept, manipulate and re-send any exchanged packet as follows:
  – It may modify any HEADER bits.
  – It may modify the Question section.
  – It may remove/add/modify any resource record, including RRSIGs, DNSKEYs, or Type A or NS records.
— It cannot access any secret cryptographic keys.
— It can only do polynomial order computations.

Since the deployment of SL-DNSSEC does not depend on the zone's parent, we omit the root (.) and the com TLD from the analysis. We also assume that a secure chain of trust exists from the root to `example.com` before SL-DNSSEC is deployed. Concretely, this secure chain of trust exists when:

(1) (.)'s $\mathsf{KSK_{pk}}$ is the trust anchor on the resolver.
(2) (.)'s $\mathsf{KSK_{sk}}$ signs (.)'s $\mathsf{ZSK_{pk}}$
(3) (.)'s $\mathsf{ZSK_{sk}}$ signs com's DS containing a hash of com's $\mathsf{KSK_{pk}}$
(4) com's $\mathsf{KSK_{sk}}$ signs com's $\mathsf{ZSK_{pk}}$
(5) com's $\mathsf{ZSK_{sk}}$ signs `example.com`'s DS containing a hash of `example.com`'s $\mathsf{KSK_{pk}}$

We now begin to scrutinize the SL-DNSSEC protocol between the resolver and the `example.com` ANS under the attacker model explicated earlier. Note that we only analyse the attack surfaces that are unique to SL-DNSSEC. Attacks also applicable to regular DNSSEC, such as modifying HEADER or unsigned glue records, have already been appraised in [13].

4.1 Attacker alters DNSKEYs sent by ANS

During phase 1 (§3.1) of the protocol, an adversary runs $\mathsf{KEM.keygen}()$ to generate its own Zone KEM Key: $(\mathsf{ZKK^{adv}_{pk}}, \mathsf{ZKK^{adv}_{sk}})$ pair. On intercepting a QTYPE DNSKEY response sent by ANS to the resolver, the adversary may do either of the following changes to the DNSKEY RRset:

— Insert $\mathsf{ZKK^{adv}_{pk}}$ into the RRset.
— Replace the authentic $\mathsf{ZKK_{pk}}$ with $\mathsf{ZKK^{adv}_{pk}}$.
— Delete $\mathsf{ZKK_{pk}}$ from the RRset.

However, assuming an EUF-CMA-secure signature scheme was used to sign the public key RRset, the RRSIG validation thereof will fail at the resolver's end.

Alternatively, the adversary generates its own Key Signing Key $(\mathsf{KSK^{adv}_{pk}}, \mathsf{KSK^{adv}_{sk}})$ pair, and substitutes $\mathsf{KSK_{pk}}$ with $\mathsf{KSK^{adv}_{pk}}$. Thereafter, it performs any of the three aforesaid amendments, and re-signs the modified RRset with $\mathsf{KSK^{adv}_{sk}}$.

This time, the resolver will successfully verify the malicious DNSKEY RRset with $\mathsf{KSK^{adv}_{pk}}$. However, assuming a *collision resistant* hash was used to compute the DS record of $\mathsf{KSK_{pk}}$, the resolver will not be able to connect the trust of $\mathsf{KSK^{adv}_{pk}}$ with the parent, thus failing to complete the full validation.

4.2 Attacker alters SL-DNSSEC query

The usage of an IND-CCA-secure KEM (refer §2.3 for the security definition) restricts the adversary in phase 2 (§3.2) to either of the manipulations underneath:

— Corrupt the ciphertext ct to ct′.
— Do a KEM encapsulation against $\mathsf{ZKK_{pk}}$ of ANS to probabilistically obtain a shared secret ($\mathsf{ss_{adv}}$) and ciphertext ($\mathsf{ct_{adv}}$).

$$(\mathsf{ss_{adv}}, \mathsf{ct_{adv}}) \xleftarrow{\$} \mathsf{KEM.encap}(\mathsf{ZKK_{pk}})$$

[8] The Signature field is opaque to middleboxes. Moreover, a MAC can be loosely thought of as a symmetric *signature* (without the non-repudiation property).

Then substitute ct with ct_{adv} in the query.

— Remove DNSKEY record holding ct from the query.

In the first case, the failure behaviour of $\text{KEM.decap}(\text{ZKK}_{\text{sk}}, \text{ct}')$ depends on the underlying KEM. In case of Kyber (refer [20], §4), if the re-encryption fails, the decapsulation will return a pseudo-random key $\text{ss}' = \text{hash}(z, \text{ct}')$, where z is a random secret seed. On the other hand, if ct′ is a valid ciphertext, the decapsulation function will return a corresponding ss′. In either case, the ANS will derive an incorrect MAC key from ss′, eventually causing MAC failure on the resolver in phase 4 (§3.4).

Concerning the second scenario, the probability that the adversary obtains a ss_{adv} such that $\text{ss}_{\text{adv}} = \text{ss}$ is negligible. Therefore, with $\text{ss}_{\text{adv}} \neq \text{ss}$, the outcome will be the same as in the first case (*i.e.* MAC verification failure).

Finally, the last attempt will convert the query to a regular one. The ANS will deem the resolver to be SL-DNSSEC-oblivious, and thus revert to a signature-based flow.

4.3 Attacker alters SL-DNSSEC response

In phase 3 (§3.3), an adversary may tamper with the DNS response in the following manner:

— Modify the resource records in any of the three sections. For example, in case of a QTYPE A response, an adversary may change the IPv4 addresses present in Type A records.

Presuming an EUF-CMA-secure MAC was used to compute authentication tags on RRsets, the resolver will fail to validate the covering RRSIGs (containing MAC tags).

5 Evaluation

5.1 Implementation

To assess the performance of SL-DNSSEC, we develop a daemon that runs on top of a DNS software (such as BIND9 or PowerDNS). Additionally, the daemon is designed to be *agnostic* to the said software (*i.e.* the underlying DNS provider can be swapped with a different one). With the daemon in place, no modifications are required to the DNS software stack, except for a small patch on the resolver's side to detect whether the Z bit in the HEADER is on/off. The Z bit is what signals to the DNS software that the response has been successfully SL-DNSSEC-validated by the daemon. We now succinctly discuss the functionality of the daemon in question.

Daemon. Figure 5 illustrates a SL-DNSSEC validation being performed with the aid of the daemon. In all our experiments, we pre-generate and hardcode the KEM keys in the daemon. In actual practice, the KEM public key and the corresponding signature thereon would be fetched by the resolver via a QTYPE DNSKEY query as discussed earlier in §3.1.

The daemon performs all SL-DNSSEC related operations independently of the DNS software (here, BIND9). Observe that the daemon on the ANS sets OPT → CLASS to 65507 (the maximum UDP payload size over IPv4) before forwarding the query to BIND. This is to allow the retrieval of the full DNS response from BIND[9] without truncation. If the size of BIND's response exceeds the resolver's (originally) advertised EDNS0 buffer size (here, 1232), the daemon replaces the signatures with MACs as outlined in §3.3.

[9] Increases in various BIND9 buffer sizes were also required.

On the resolver, the daemon performs the SL-DNSSEC validation of DNS records (as elucidated in §3.4) and sets HEADER → Z = 1 in case of a successful outcome.

Software Setup. We use the source code of QBF [57] as base to build the SL-DNSSEC daemon. The DNS software is a BIND 9.19.17 fork [7] which supports NIST level I PQC signatures. In the fork, we further add support for:

(1) NIST level V Falcon and Dilithium schemes
(2) Detecting the Z bit in the HEADER

The cryptographic stack is `openssl` 3.2, `liboqs` 0.10.0 [67] and `oqs-provider` 0.6.0. The daemon is written in C and uses the library `libnetfilter-queue` to intercept incoming and outgoing DNS packets. Docker 4.29 is used for constructing the network scenario (described below). To simulate network bandwidth and latency, we use Linux's `tc` utility. DNS queries are issued using `dig`. Communication statistics are obtained with `ip` command. All experiments are run on a MacBook Air M1 with 8 GB of RAM.

Network Scenario. The DNS network contains the following four participants: 1) A client 2) A resolver 3) A root (`.`) nameserver 4) An `example` authoritative nameserver (ANS). We skip configuring a `com` TLD to reduce complexity. Each participant runs as a private Ubuntu 22.04 Docker container with experiment-specific bandwidth and latency constraints. Additionally, the SL-DNSSEC daemon is installed on both the resolver and the ANS containers.

The EDNS0 buffer size is set to the recommended value of 1232. For simplicity, each zone is signed with a single algorithm and has one ZSK and one KSK. The daemons on both the resolver and the ANS are also pre-configured with the requisite KEM ZKK keys.

The zone file served by the ANS contains 10 Type A records, each with a unique domain name and an associated RRSIG. The ANS is configured with `minimal-responses no-auth-recursive;` (the default setting that ships with BIND) which means that it will be *as complete as possible* while generating responses for iterative queries. Such a response is called *non-minimal* and represents the worst-case scenario in terms of message size. Refer §3.3 for the number and the type of records contained in a *non-minimal* QTYPE A response returned by the ANS in the described setup. To facilitate modifications to DNS messages without readjusting compression name pointers, we also set `message-compression no;` in `named.conf`.

5.2 Experiments and Results

We now assess SL-DNSSEC's performance against signature-based DNSSEC in terms of bandwidth usage and resolution times. We conduct two experiments targeting NIST security level I and V[10], respectively. Before the start of an experiment, the resolver pre-fetches DNSKEY and NS records of all the zones, including the DS record of `example` zone. The implication is that the resolver directly contacts the ANS in order to resolve the client's query, rather than starting the lookup process all the way up from the root.

Signature-based DNSSEC instances are run over two transports:

(1) Standard DNS (SD) over UDP with a fallback to TCP in case of a truncated (TC) response
(2) An upper-layer UDP-only fragmentation scheme such as ARRF/QBF

[10] NIST level V experimental results are in the Appendix.

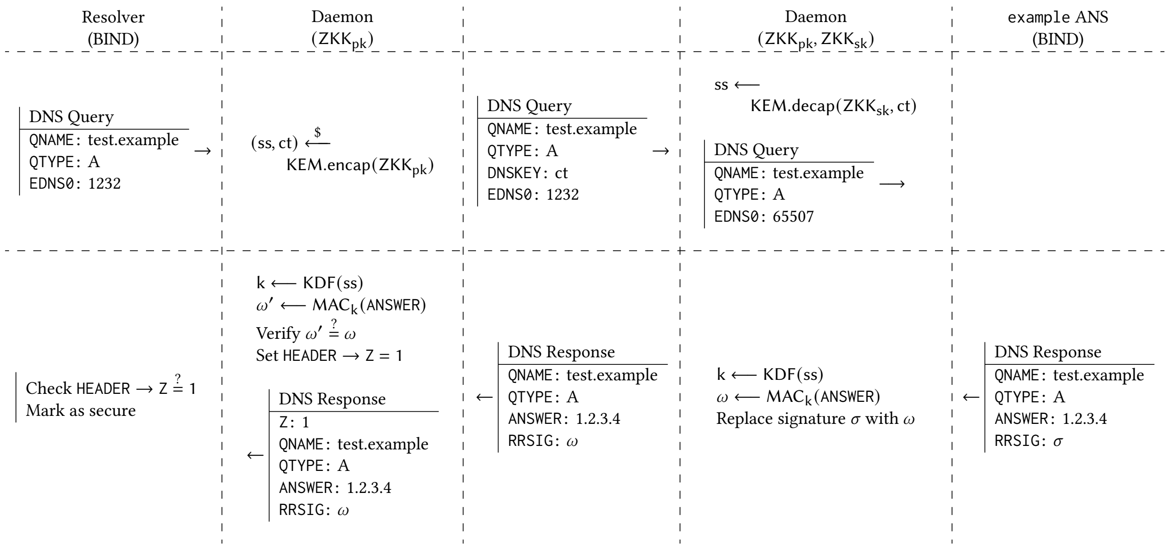
Resolver (BIND)
Daemon (ZKK_{pk})
Daemon (ZKK_{pk}, ZKK_{sk})
example ANS (BIND)
DNS Query
QNAME: test.example
QTYPE: A
EDNS0: 1232
(ss, ct) $\xleftarrow{\$}$ KEM.encap(ZKK_{pk})
DNSKEY: ct
ss ⟵ KEM.decap(ZKK_{sk}, ct)
EDNS0: 65507
k ⟵ KDF(ss)
ω' ⟵ MAC_k(ANSWER)
Verify $\omega' \overset{?}{=} \omega$
Set HEADER → Z = 1
Check HEADER → Z $\overset{?}{=}$ 1
Mark as secure
DNS Response
Z: 1
ANSWER: 1.2.3.4
RRSIG: ω
ω ⟵ MAC_k(ANSWER)
Replace signature σ with ω
RRSIG: σ

Figure 5: An overview of SL-DNSSEC validation via Daemon (NIST Level I)

We exclude ARRF from our experiments as its performance can be easily extrapolated from that of QBF. Both schemes (since they mainly differ in the way fragments are packaged) have roughly the same bandwidth usage, with QBF being a round-trip faster than ARRF. The number of packets sent and received also remains within ±1 margin, respectively.

Each experiment consists of two main stages: 1) Measure the bandwidth consumption during a single query resolution, and 2) Measure the mean resolution time of 10 queries.

Measuring bandwidth usage. We send a QTYPE A DNS query from the client to the resolver. At the resolver's Ethernet interface, we then assess the communication with the ANS in terms of:

- Number of packets (technically, frames) in transmit (TX) and receive (RX)
- Number of bytes in transmit (TX) and receive (RX). Note that these values include:
  1. 14-byte Ethernet header
  2. 20-byte IPv4 header
  3. 8-byte UDP header or 32-byte TCP header (40-byte in case of SYN and SYN-ACK)
- Transport protocol used. Here, TCP* indicates a TCP fallback wherein the first round-trip is over UDP while the subsequent ones are over TCP. In this case, the exchanged bytes also include SYN, ACK and FIN packets.

Measuring resolution time. We measure the DNS query resolution speed, with each participant configured with the following networking capabilities[11]:

1. High Bandwidth (100 Mbps), Low Latency (10 ms)
2. Low Bandwidth (1 Mbps), High Latency (100 ms)

Specifically, we issue 10 QTYPE A DNS queries from the client to the resolver and calculate the mean resolution time. That is, the average time elapsed between the client sending its query and subsequently receiving a DNSSEC validated response (with HEADER → AD set) from the resolver.

[11] Latency being one-way, RTT = 2× Latency. Moreover, packet loss rate = 0%. This is the best-case performance scenario for fragmentation schemes (ARRF/QBF).

5.2.1 *Experiment 1.*

We target NIST level I parameters. To instantiate SL-DNSSEC, we use the following primitives:

- **Post-Quantum KEM:** Kyber-512
- **KDF:** HKDF-SHA-256 [44]
- **MAC:** HMAC-SHA-256 [43]

To determine how SL-DNSSEC fares against signature-based DNSSEC, we sign the zone file with the schemes below:

– Pre-Quantum: RSA-2048, ECDSA P-256
– Post-Quantum: Falcon-512, Dilithium-2, SPHINCS$^+$-128s

Results and Discussion. All the results of the experiment are catalogued in Tables (11, 12).

We observe that the Kyber-HMAC instance of SL-DNSSEC, while additionally providing a level I post-quantum security, is virtually equivalent to RSA-2048 in terms of total bytes exchanged and resolution times. In fact, out of all the tested mechanisms, Kyber-HMAC has the smallest response size (RX), even beating out ECDSA-SD.

Concerning NIST selected signatures, Kyber-HMAC requires less than half the bandwidth of Falcon instances. On bringing Dilithium and SPHINCS$^+$ into the picture, the bandwidth savings become even more dramatic (*i.e.* 86% and 95%, respectively). The upshot is that, with SL-DNSSEC, servers will not need to upgrade to a higher bandwidth connection, thus shrinking the operational costs.

Moreover, Kyber-HMAC (SL-DNSSEC) remains immune against being exploited as a DNS amplifier. Observe that Kyber-HMAC transmits a large query[12] owing to the KEM ciphertext contained therein. Due to a response being *smaller* than its query, the Bandwidth Amplification Factor (BAF) becomes < 1, resulting in a negative return on bandwidth investment for a prospective attacker. On the other hand. all signature-based methods over UDP can be potentially exploited by an attacker for DDoS amplifications attacks.

[12] The *total* bandwidth usage still remains comparable to RSA-2048.

Table 11: A comparison of resolver's bandwidth usage. SD denotes Standard DNS. TCP* : TCP fallback. (NIST Level I)

Method	Via	Pkts. Sent TX	Pkts. Rcvd. RX	Bytes Sent TX	Bytes Rcvd. RX	BAF	Bytes Xferred TX+RX
ECDSA-SD	UDP	1	1	84	512	6.1	596
RSA-SD	UDP	1	1	84	1088	13	1172
Falcon-SD	TCP*	8	6	598	2700	-	3298
Dilithium-SD	TCP*	13	11	928	8292	-	9220
SPHINCS$^+$-SD	TCP*	24	24	1654	25389	-	27043
Falcon-QBF	UDP	3	3	258	2788	10.8	3046
Dilithium-QBF	UDP	8	8	693	9225	13.3	9918
SPHINCS$^+$-QBF	UDP	23	23	2012	28321	14.1	30333
Kyber-HMAC	UDP	1	1	867	416	0.48	1283

Table 12: A comparison of client's query resolution time. SD denotes Standard DNS. (NIST Level I)

Method	100 Mbps, 10 ms Avg. Resolution Time (±1 ms)	1 Mbps, 100 ms Avg. Resolution Time (±2 ms)
ECDSA-SD	44	407
RSA-SD	44	408
Falcon-SD	89	811
Dilithium-SD	89	817
SPHINCS$^+$-SD	111	1025
Falcon-QBF	45	410
Dilithium-QBF	46	415
SPHINCS$^+$-QBF	48	436
Kyber-HMAC	44	408

With reference to post-quantum signatures over Standard DNS (SD), we observe a slowdown of at least 50% compared to other setups. This because the DNS response containing PQC signatures always exceeds the EDNS0 buffer size of 1232 bytes. Consequently, the initial UDP round-trip is wasted (due to the response being marked truncated (TC)) and overall resolution times further increased (due to the ensuing 3-way TCP handshake).

Interestingly, SPHINCS$^+$-SD even incurs an extra round-trip compared to Falcon-SD and Dilithium-SD. This is because the size of a SPHINCS$^+$ QTYPE A response exceeds the Initial Congestion Window (`initcwnd`) of 10 segments set in the TCP slow start algorithm [18, 22]. Given the default Maximum Segment Size (MSS) of 1220 bytes, the size of `initcwnd` comes out to be $10 \times 1220 = 12.2$ KB. The repercussion of exceeding this `initcwnd` is that after sending about 12.2 KB of data, the nameserver waits for the resolver to acknowledge (ACK) the received packets, before continuing with the rest of the transmission.

While QBF matches the resolution speeds of classical DNSSEC, it exchanges multiple DNS/UDP packets in proportion to the size of the original (un-fragmented) response. Given the absence of any flow and congestion control mechanisms in UDP, this torrent of packets can potentially exhaust the network resources of busy resolvers/nameservers and overwhelm middleboxes, whilst also increasing the chances of the session requiring more round-trips due to unanticipated packet drops.

Surprisingly, RSA-SD has a higher BAF than Falcon-QBF, despite Falcon's signature being almost 2.6× the size of RSA's. This is because QBF sends multiple full-fledged DNS queries, which increases the amount of TX bytes (the denominator), thereby amortizing BAF.

Finally, observe that QBF starts to consume slightly more bandwidth than Standard DNS (SD) as the number of exchanged packets grow. This is because QBF first envelops a signature fragment in an RRSIG record. This RRSIG is then inserted in a DNS message (along with its concomitant 12-byte HEADER, Question section, OPT record, etc.) resulting in a data overhead. TCP, on the other hand, is a continuous byte-stream of the original DNS response.

6 Conclusion

We presented the SL-DNSSEC protocol: a backward-compatible and signature-*free* alternative for performing DNSSEC validations in a single UDP query/response fashion. Leveraging a quantum-safe KEM and a MAC, SL-DNSSEC achieves NIST level I security while having analogous bandwidth usage and query resolution speeds to that of RSA-2048. DNS messages in SL-DNSSEC remain below the recommended 1232-byte size, thus avoiding IP fragmentation or TCP re-transmission. Moreover, owing to smaller responses than queries, SL-DNSSEC cannot be misused as a DNS amplifier.

Availability

The software artifact germane to this work is available online at https://github.com/aditya-asr/sl-dnssec.

Acknowledgments

We thank the anonymous reviewers for their helpful suggestions. The co-author of this work, M.P. Jhanwar, would also like to acknowledge the MPhasis F1 Foundation for their research grant.

References

[1] [n. d.]. Dan Kaminsky, Black Ops 2008: It's The End Of The Cache As We Know It. https://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Kaminsky/BlackHat-Japan-08-Kaminsky-DNS08-BlackOps.pdf. Accessed: 2024-07-09.
[2] [n. d.]. The DDoS That Almost Broke the Internet. https://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet. Accessed: 2024-07-09.
[3] [n. d.]. DNS Flag Day 2020. https://www.dnsflagday.net/2020/.
[4] [n. d.]. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/.
[5] [n. d.]. Is large-scale DNS over TCP practical? https://ripe76.ripe.net/presentations/95-jonglez-dns-tcp-ripe76.pdf. Accessed: 2024-07-09.
[6] [n. d.]. Noise Protocol Framework. https://noiseprotocol.org/noise.html.
[7] [n. d.]. OQS-bind. https://github.com/Martyrshot/OQS-bind.
[8] [n. d.]. Subdomain enumeration with DNSSEC. https://blog.apnic.net/2023/01/17/subdomain-enumeration-with-dnssec/. Accessed: 2025-01-03.
[9] 2020. Defragmenting DNS - Determining the optimal maximum UDP response size for DNS. https://indico.dns-oarc.net/event/36/contributions/776/ Accessed: 2024-07-09.
[10] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J. Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, Seth Schoen, and Brad Warren. 2019. Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web. In *ACM CCS*.
[11] Suranjith Ariyapperuma and Chris J. Mitchell. 2007. Security vulnerabilities in DNS and DNSSEC. In *ARES*.
[12] Derek Atkins and Rob Austein. 2004. Threat Analysis of the Domain Name System (DNS). RFC 3833.

[13] Jason Bau and John C. Mitchell. 2010. A Security Evaluation of DNSSEC with NSEC3. In *NDSS*.

[14] G.J. Beernink. 2022. *Taking the quantum leap: Preparing DNSSEC for Post Quantum Cryptography*. Master's thesis. University of Twente. http://essay.utwente.nl/89509/

[15] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. 1998. A modular approach to the design and analysis of authentication and key exchange protocols. In *STOC*.

[16] Mihir Bellare and Phillip Rogaway. 1994. Entity Authentication and Key Distribution. In *CRYPTO*.

[17] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. 2019. The SPHINCS+ Signature Framework. In *ACM CCS*.

[18] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. 2009. TCP Congestion Control. RFC 5681.

[19] Ron Bonica, Fred Baker, Geoff Huston, Bob Hinden, Ole Trøan, and Fernando Gont. 2020. IP Fragmentation Considered Fragile. RFC 8900.

[20] Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. 2018. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *EuroS&P*.

[21] Ran Canetti and Hugo Krawczyk. 2001. Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels. In *EUROCRYPT*.

[22] Jerry Chu, Nandita Dukkipati, Yuchung Cheng, and Matt Mathis. 2013. Increasing TCP's Initial Window. RFC 6928.

[23] Taejoong Chung, Roland Van Rijswijk-Deij, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. 2017. A longitudinal, end-to-end view of the DNSSEC ecosystem. In *USENIX*.

[24] Joao da Silva Damas, Michael Graff, and Paul A. Vixie. 2013. Extension Mechanisms for DNS (EDNS(0)). RFC 6891.

[25] Cyprien de Saint Guilhem, Nigel P. Smart, and Bogdan Warinschi. 2017. Generic Forward-Secure Key Agreement Without Signatures. In *ISC*.

[26] Pratyush Dikshit, Mike Kosek, Nils Faulhaber, Jayasree Sengupta, and Vaibhav Bajpai. 2024. Evaluating DNS Resiliency and Responsiveness With Truncation, Fragmentation & DoTCP Fallback. *IEEE TNSM* (2024).

[27] Jason A. Donenfeld. 2017. WireGuard: Next Generation Kernel Network Tunnel. In *NDSS*.

[28] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR TCHES* (2018).

[29] Donald E. Eastlake and Mark P. Andrews. 2016. Domain Name System (DNS) Cookies. RFC 7873.

[30] Andrew Fregly, Joseph Harvey, Burton S. Kaliski Jr., and Swapneel Sheth. 2023. Merkle Tree Ladder Mode: Reducing the Size Impact of NIST PQC Signature Algorithms in Practice. In *CT-RSA*.

[31] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. 2012. Strongly Secure Authenticated Key Exchange from Factoring, Codes, and Lattices. In *PKC*.

[32] Jason Goertzen and Douglas Stebila. 2023. Post-Quantum Signatures in DNSSEC via Request-Based Fragmentation. In *PQCrypto*.

[33] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *STOC*.

[34] Amir Herzberg and Haya Shulman. 2013. Fragmentation Considered Poisonous, or: One-domain-to-rule-them-all.org. In *IEEE CNS*.

[35] Paul E. Hoffman and Patrick McManus. 2018. DNS Queries over HTTPS (DoH). RFC 8484.

[36] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and Paul E. Hoffman. 2016. Specification for DNS over Transport Layer Security (TLS). RFC 7858.

[37] Christian Huitema, Sara Dickinson, and Allison Mankin. 2022. DNS over Dedicated QUIC Connections. RFC 9250.

[38] Shumon Huque, Christian Elmerot, and Ólafur Guðmundsson. 2024. *Compact Denial of Existence in DNSSEC.* Technical Report draft-ietf-dnsop-compact-denial-of-existence-05.

[39] Philipp Jeitner and Haya Shulman. 2021. Injection Attacks Reloaded: Tunnelling Malicious Payloads over DNS. In *USENIX*.

[40] Panos Kampanakis and Tancrède Lepoint. 2023. Vision Paper: Do We Need to Change Some Things?. In *SSR*.

[41] Charlie Kaufman, Radia Perlman, and Bill Sommerfeld. 2003. DoS protection for UDP-based protocols. In *ACM CCS*.

[42] Mike Kosek, Trinh Viet Doan, Simon Huber, and Vaibhav Bajpai. 2022. Measuring DNS over TCP in the era of increasing DNS response sizes: a view from the edge. *ACM CCR* (2022).

[43] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. 1997. HMAC: Keyed-Hashing for Message Authentication. RFC 2104.

[44] Dr. Hugo Krawczyk and Pasi Eronen. 2010. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869.

[45] H. Krawczyk. 1996. SKEME: a versatile secure key exchange mechanism for Internet. In *NDSS*.

[46] Hugo Krawczyk. 2005. HMQV: A High-Performance Secure Diffie-Hellman Protocol. In *CRYPTO*.

[47] Hugo Krawczyk and Hoeteck Wee. 2016. The OPTLS Protocol and TLS 1.3. In *EuroS&P*.

[48] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. 2007. Stronger Security of Authenticated Key Exchange. In *ProvSec*.

[49] Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone. 2003. An Efficient Protocol for Authenticated Key Agreement. *DCC* (2003).

[50] Matt Lepinski and Stephen Kent. 2012. An Infrastructure to Support Secure Internet Routing. RFC 6480.

[51] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. 2020. DNS Cache Poisoning Attack Reloaded: Revolutions with Side Channels. In *ACM CCS*.

[52] Keyu Man, Xin'an Zhou, and Zhiyun Qian. 2021. DNS Cache Poisoning Attack: Resurrections with Side Channels. In *ACM CCS*.

[53] Jiarun Mao, Michael Rabinovich, and Kyle Schomp. 2022. Assessing Support for DNS-over-TCP in the Wild. In *PAM*.

[54] Giovane C. M. Moura, Moritz Müller, Marco Davids, Maarten Wullink, and Cristian Hesselman. 2021. Fragmentation, Truncation, and Timeouts: Are Large DNS Messages Falling to Bits?. In *PAM*.

[55] Moritz Müller, Jins de Jong, Maran van Heesch, Benno Overeinder, and Roland van Rijswijk-Deij. 2020. Retrofitting post-quantum cryptography in internet protocols: a case study of DNSSEC. *ACM CCR* (2020).

[56] T. Prest, P.A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z Zhang. 2022. FALCON. Tech. rep., National Institute of Standards and Technology, available at. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[57] Aditya Singh Rawat and Mahabir Prasad Jhanwar. 2023. Post-quantum DNSSEC over UDP via QNAME-Based Fragmentation. In *SPACE*.

[58] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. 2005. DNS Security Introduction and Requirements. RFC 4033.

[59] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. 2005. Protocol Modifications for the DNS Security Extensions. RFC 4035.

[60] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. 2005. Resource Records for the DNS Security Extensions. RFC 4034.

[61] Christian Rossow. 2014. Amplification Hell: Revisiting Network Protocols for DDoS Abuse.. In *NDSS*.

[62] Peter Schwabe, Douglas Stebila, and Thom Wiggers. 2020. Post-Quantum TLS Without Handshake Signatures. In *ACM CCS*.

[63] Peter Schwabe, Douglas Stebila, and Thom Wiggers. 2021. More Efficient Post-quantum KEMTLS with Pre-distributed Public Keys. In *ESORICS*.

[64] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SICOMP* (1997).

[65] Mukund Sivaraman, Shane Kerr, and Linjian Song. [n. d.]. DNS message fragments. https://datatracker.ietf.org/doc/draft-muks-dns-message-fragments/00/.

[66] Linjian Song and Shengling Wang. [n. d.]. ATR: Additional Truncation Response for Large DNS Response. https://datatracker.ietf.org/doc/draft-song-atr-large-resp/03/.

[67] Douglas Stebila and Michele Mosca. 2017. Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project. In *SAC*.

[68] Gijs Van Den Broek, Roland Van Rijswijk-Deij, Anna Sperotto, and Aiko Pras. 2014. DNSSEC meets real world: dealing with unreachability caused by fragmentation. *IEEE Communications Magazine* (2014).

[69] Roland van Rijswijk-Deij, Mattijs Jonker, Anna Sperotto, and Aiko Pras. 2016. A High-Performance, Scalable Infrastructure for Large-Scale Active DNS Measurements. *IEEE JSAC* (2016).

[70] Roland van Rijswijk-Deij, Anna Sperotto, and Aiko Pras. 2014. DNSSEC and its potential for DDoS attacks: a comprehensive measurement study. In *IMC*.

[71] Sam Weiler and Johan Stenstam. 2006. Minimally Covering NSEC Records and DNSSEC On-line Signing. RFC 4470.

[72] Thom Wiggers. 2024. *Post-Quantum TLS.* Ph. D. Dissertation. Radboud University. https://thomwiggers.nl/publication/thesis/

A Other Considerations

A.1 Fetching DNSKEYs from Nameservers

Note that SL-DNSSEC relies upon a PQC signature on the DNSKEY RRset to ascertain the authenticity of the KEM public key ($\mathsf{ZKK}_{\mathsf{pk}}$) contained therein. Although DNS responses in SL-DNSSEC remain well under 1232 bytes, the initial QTYPE `DNSKEY` response from a nameserver containing multiple DNSKEYs and one or more RRSIGs will likely not respect the aforesaid size ceiling.

Therefore, similar to regular DNSSEC, retrieving the DNSKEYs of a zone may entail the use of either one of the following methods:

(1) An upper-layer fragmentation scheme (ARRF/QBF)
(2) Standard DNS with a fallback to TCP
(3) An out-of-band distribution via HTTP or FTP

Fortunately, this is not much of a concern since DNSKEYs are fetched infrequently owing to their higher caching TTLs.

A.2 Managing Keys on Nameservers

Considerations that are pertinent to DNSSEC in live (on-the-fly) signing mode also remain applicable to SL-DNSSEC. Specifically, since KEM keys will be stored on nameservers that connect to the Internet (which increases the overall attack surface), a hardware security module (HSM) is therefore recommended for a secure management of the keys.

A.3 Computational Requirements

Table 13 contrasts the computations performed in SL-DNSSEC and DNSSEC in online/offline signing mode. We also benchmark[13] various algorithms in a docker container running on a MacBook Pro M4 with 16 GB RAM. The results are catalogued in Table 14.

We now give a heuristic analysis of SL-DNSSEC's compute requirements. First, note that HMAC and HKDF operations pose a minor overhead on DNS servers for the following reasons:

(1) HKDF internally uses HMAC, which in turn uses hashing. Thus, both primitives use hashing as their core operation. On modern hardware, hashing is extremely fast.
(2) DNS servers are already hashing-capable since they compute DNS cookies [29] (which are usually HMACs).

Next, note that Cloudflare has successfully deployed DNSSEC live signing at scale on their servers with ECDSA P-256[14]. Since Kyber operations are much faster than ECDSA's (refer Table 14), SL-DNSSEC can be straightforwardly deployed on online signing servers, likely with major efficiency gains.

In comparison to DNSSEC in offline signing mode, SL-DNSSEC does introduce a KEM decapsulation operation on nameservers. We remark, however, that there is a growing incentive to switch DNSSEC from offline to online computation, even at the cost of some overhead. This is because `NSEC(3)` *zone walking*[15] [8] countermeasures, such as white or black lies [38, 71], require on-the-fly generation of signatures (or MACs in case of SL-DNSSEC). Another benefit of live computation is the ability to generate records dynamically (*e.g.*, geolocation-based answers).

Finally, SL-DNSSEC resolvers are expected to have significantly reduced CPU load since KEM encapsulations are computationally far less intensive than signature verifications.

B Experiment 2 (NIST Level V)

To assess SL-DNSSEC's scalability, we target NIST level V. This is the highest security level and is *likely excessive* for DNSSEC [14].

Choice of Primitives. The updated parameters are:

- **Post-Quantum KEM:** Kyber-1024
- **KDF:** HKDF-SHA-512 [44]
- **MAC:** HMAC-SHA-512 [43]

To compare SL-DNSSEC with signature-based DNSSEC, we sign the zone file with Falcon-1024 and Dilithium-5. Table 15 compares the object sizes of various NIST level V signature and KEM schemes. We omit testing SPHINCS$^+$ since the resulting response would exceed 64 KB, the maximum possible size for a DNS message. Furthermore, since a DNS query carrying a Kyber ciphertext of 1568 bytes would exceed the recommended threshold of 1232, we adapt QBF [57] to split ct into two DNS queries, as sketched in Fig. 6.

Results and Discussion. All the findings of Experiment 2 are rendered in Tables (16, 17). Compared to Falcon and Dilithium

[13] Using `openssl speed` and `speed_(kem/sig)` of `liboqs`.
[14] RSA-2048 is not used because of its low signing throughput.
[15] This attack is used to retrieve the entire contents of a DNS zone.

Table 13: A comparison of cryptographic operations

	SL-DNSSEC	**DNSSEC Live Signing**	**DNSSEC Offline Signing**
Resolver Query	KEM encap	-	-
Nameserver Response	KEM decap KDF MAC	SIG sign	-
Resolver Validation	KDF MAC	SIG verify	SIG verify

Table 14: Computational speeds of various algorithms

Primitive	**Operations/second**
HMAC-SHA256	8398588 hmac/s
RSA-2048	2725 sign/s 104064 verify/s
ECDSA P-256	82027 sign/s 27312 verify/s
Falcon-512	7721 sign/s 61663 verify/s
Dilithium-2	13252 sign/s 37417 verify/s
SPHINCS$^+$-128s	5 sign/s 4544 verify/s
Kyber-512	130503 encap/s 150779 decap/s

Table 15: A size comparison (in bytes) of signature (sig) / ciphertext (ct) and public key (pk) of various algorithms.

Algorithm	**sig / ct**	**pk**
Falcon-1024	1280	1793
Dilithium-5	4595	2592
SPHINCS$^+$-256s	29792	64
Kyber-1024	1568	1568

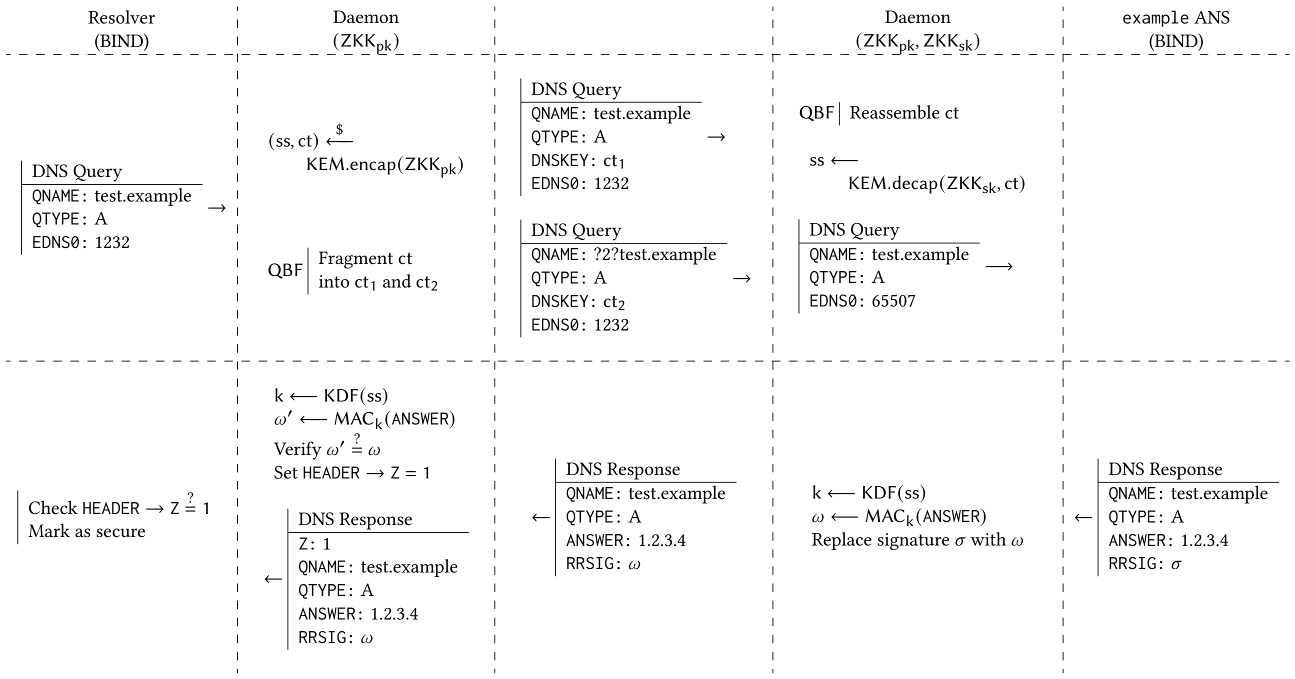
Resolver (BIND)
Daemon ($\mathsf{ZKK_{pk}}$)
Daemon ($\mathsf{ZKK_{pk}}$, $\mathsf{ZKK_{sk}}$)
example ANS (BIND)
DNS Query
QNAME: test.example
QTYPE: A
EDNS0: 1232
$(\mathrm{ss}, \mathrm{ct}) \xleftarrow{\$} \mathsf{KEM.encap}(\mathsf{ZKK_{pk}})$
QBF | Fragment ct into ct_1 and ct_2
DNS Query
QNAME: test.example
QTYPE: A
DNSKEY: ct_1
EDNS0: 1232
DNS Query
QNAME: ?2?test.example
QTYPE: A
DNSKEY: ct_2
EDNS0: 1232
QBF | Reassemble ct
$\mathrm{ss} \longleftarrow \mathsf{KEM.decap}(\mathsf{ZKK_{sk}}, \mathrm{ct})$
DNS Query
QNAME: test.example
QTYPE: A
EDNS0: 65507
$\mathrm{k} \longleftarrow \mathsf{KDF}(\mathrm{ss})$
$\omega' \longleftarrow \mathsf{MAC_k}(\mathsf{ANSWER})$
Verify $\omega' \overset{?}{=} \omega$
Set HEADER $\rightarrow$ Z = 1
Check HEADER $\rightarrow$ Z $\overset{?}{=}$ 1
Mark as secure
DNS Response
Z: 1
QNAME: test.example
QTYPE: A
ANSWER: 1.2.3.4
RRSIG: ω
DNS Response
QNAME: test.example
QTYPE: A
ANSWER: 1.2.3.4
RRSIG: ω
$\mathrm{k} \longleftarrow \mathsf{KDF}(\mathrm{ss})$
$\omega \longleftarrow \mathsf{MAC_k}(\mathsf{ANSWER})$
Replace signature σ with ω
DNS Response
QNAME: test.example
QTYPE: A
ANSWER: 1.2.3.4
RRSIG: σ

Figure 6: An overview of SL-DNSSEC validation via Daemon (NIST Level V)

Table 17: A comparison of client's query resolution time. SD denotes Standard DNS. (NIST Level V)

Method	100 Mbps, 10 ms Avg. Resolution Time (±1 ms)	1 Mbps, 100 ms Avg. Resolution Time (±2 ms)
Falcon-SD	89	812
Dilithium-SD	111	1014
Falcon-QBF	45	411
Dilithium-QBF	47	426
Kyber-HMAC	44	409

Table 16: A comparison of resolver's bandwidth usage. SD : Standard DNS. TCP* : TCP fallback. (NIST Level V)

Method	Via	Pkts. Sent TX	Pkts. Rcvd. RX	Bytes Sent TX	Bytes Rcvd. RX	BAF	Bytes Xferred TX+RX
Falcon-SD	TCP*	10	8	730	4674	-	5404
Dilithium-SD	TCP*	18	16	1258	15147	-	16405
Falcon-QBF	UDP	4	4	345	4865	14.1	5210
Dilithium-QBF	UDP	14	14	1220	17165	14.1	18385
Kyber-HMAC	UDP	2	1	1766	512	0.29	2278

instances, Kyber-HMAC (SL-DNSSEC) still manages to cut bandwidth consumption by about 56% and 86%, respectively. Furthermore, thanks to small MACs, the response size (RX) in Kyber-HMAC increases by only 96 bytes despite the big jump in security level. However, to meet the UDP size constraints of 1232 bytes, the resolver daemon unavoidably has to dispatch an extra query.

Lastly, while Falcon-SD remains consistent with its resolution speeds, Dilithium-SD suffers a penalty of an extra round-trip because of exceeding TCP's `initcwnd`. Notice that the BAF in Falcon-QBF and Dilithium-QBF is the same, despite a large discrepancy in the underlying signature sizes. Again, this is to be ascribed to the higher amount of TX bytes (due to more DNS queries) in the latter.