Post-Quantum DNSSEC with Faster TCP Fallbacks

Aditya Singh Rawat and Mahabir Prasad Jhanwar*

Ashoka University, Sonipat, India
`{aditya.rawat_phd21,mahavir.jhawar}@ashoka.edu.in`

Abstract. In classical DNSSEC, a drop-in replacement with quantum-safe cryptography would increase DNS query resolution times by *at least* a factor of 2×. Since a DNS response containing large post-quantum signatures is likely to get marked truncated (`TC`) by a nameserver (resulting in a wasted UDP round-trip), the client (here, the resolver) would have to retry its query over TCP, further incurring a *minimum* of two round-trips due to the three-way TCP handshake.
We present TurboDNS: a backward-compatible protocol that eliminates *two* round-trips from the preceding flow by 1) sending TCP handshake data in the initial DNS/UDP flight itself, and 2) immediately streaming the DNS response over TCP after authenticating the client with a cryptographic cookie. Our experiments show that DNSSEC over TurboDNS, with either Falcon-512 or Dilithium-2 as the zone signing algorithm, is practically as fast as the currently deployed ECDSA P-256 and RSA-2048 setups in resolving `QTYPE A` DNS queries.

Keywords: Network Security · Post-quantum Cryptography Implementations · DNSSEC

1 Introduction

A quantum computer running Shor's period finding algorithm [39] is capable of solving the factoring and the discrete logarithm problem (DLP) in polynomial time. Public-key cryptosystems (such as RSA and ECDSA), which base their security on the preceding assumptions, will therefore need to be replaced with their quantum-safe counterparts in the imminent future.

Many modern protocols deployed over the Internet, such as SSH and TLS, rely on asymmetric cryptography for their security. The DNS Security Extensions (DNSSEC) [35,37,36], being one such protocol, employs digital signatures for validation of DNS responses. Being a mission-critical service, the Domain Name System (DNS) facilitates the navigation of the Internet by translating a human-readable domain name (`www.example.com`) to a machine-understandable IP address (`1.2.3.4`). Nowadays, DNS services are also used for email authentication [21], acquisition of TLS certificates by proving a domain's ownership [5], and supporting Internet routing security (RPKI) [26,13].

*M.P. Jhanwar acknowledges the MPhasis F1 Foundation for their research grant.

Without DNSSEC in place, DNS remains vulnerable to various cache poisoning attacks [8,7,22]. An off-path adversary who can guess the 16-bit UDP[1] source port and the 16-bit transaction `ID` of a DNS query can inject a false domain-to-IP entry into a resolver's cache, thereby redirecting the users of the *poisoned* resolver to a malicious website. Recently, researchers [27,28] even found critical vulnerabilities in DNS software stacks that shrunk this search space from 2^{32} to 2^{17}, effectively enabling them to compromise resolvers' caches.

With the quantum era on the horizon, the National Institute of Standards and Technology (NIST) recently selected Crystals-Kyber [12] as Key Encapsulation Mechanism (KEM), and Falcon [32], Crystals-Dilithium [16] and SPHINCS$^+$ [9] as digital signature schemes. In comparison to their classical siblings however, these algorithms (colloquially referred to under the acronym of PQC *i.e.* Post-Quantum Cryptography), have markedly larger public key and signature sizes as demonstrated in Table 1 below.

Table 1. A comparison of public key (pk) and signature (sig) sizes in bytes

Algorithm	**Assumption**	**Level**	**Quantum-safe**	**pk**	**sig**
ECDSA-P256	ECDLP	-	✗	64	64
RSA-2048	Factoring	-	✗	260	256
Falcon-512	Lattice	I	✓	897	666
Dilithium-2	Lattice	II	✓	1312	2420
SPHINCS$^+$-128s	Hash	I	✓	32	7856
Falcon-1024	Lattice	V	✓	1793	1280
Dilithium-5	Lattice	V	✓	2592	4595
SPHINCS$^+$-256s	Hash	V	✓	64	29792

1.1 DNS Size Constraints

The sizeable footprint of PQC, as catalogued above, will have major implications for the global DNS infrastructure. A DNS over UDP message, as originally standardized, was restricted to a maximum size of 512 bytes. Bearing in mind the headroom required by DNSSEC (for conveying signatures and public keys), this size ceiling was later increased to a *theoretical* value of 64 KB with Extension Mechanisms for DNS (EDNS0) [40] in 2013.

Unfortunately, a DNS packet exceeding the Path MTU (Maximum Transmission Unit), which is typically 1500 bytes (<< 64 KB), triggers IP fragmentation at the intermediary routers. The resulting UDP/IP fragments not only may never arrive [44,11] due to being blocked by stateless firewalls but also can be used to exhaust a resolver's resources [24] or to inject spoofed records in a DNS response [19]. Additionally, the survey of [44] has revealed that approximately 10% of resolvers fail to handle IP fragments correctly.

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

In order to avoid the fragility of network layer fragmentation, DNS messages are recommended to be at most **1232** bytes in size [4,31]. This conservative limit, derived as 1280 (IPv6 minimum MTU) − 40 (IPv6 Header) − 8 (UDP Header), is deemed to obviate IP fragmentation on typical network links [3,30].

For conveying DNS messages larger than 1232 bytes, there are currently two mechanisms available: 1) TCP fallback and 2) Application layer fragmentation.

TCP Fallback. In the Standard DNS flow, a response is marked as truncated (`TC`-bit 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 BIND (a 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.

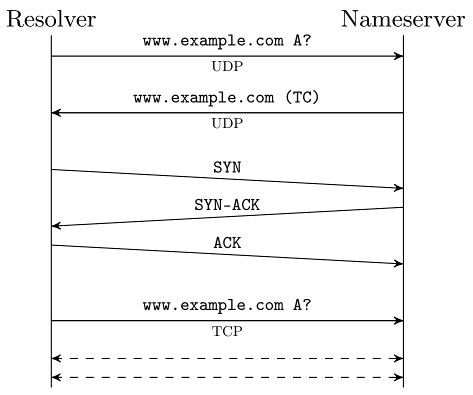

Fig. 1. Standard DNS with TCP Fallback

Application Layer Fragmentation. In this approach, fragmentation is performed at the application (DNS) layer, thereby circumventing network layer fragmentation. A large DNS message is split across several fragments, such that the size of each fragment is $\leq$ 1232 bytes. Each fragment is then sent over UDP. Thus, the nameserver becomes responsible for the fragmentation of a DNS response and the resolver for the subsequent reassembly thereof. Modern upper-layer fragmentation schemes are *request*-based in that each fragment has to be explicitly queried for. Currently, there are two such schemes available:

1) ARRF [18]. When a DNS response exceeds the UDP limits, the resource records contained therein are fragmented and sent (upon request) in pseudo-records of Type `RRFRAG`. Unfortunately, `RRFRAG` is a non-standard record Type which could cause middleboxes to drop the message. ARRF is also vulnerable to memory exhaustion attacks as acknowledged by its authors in [18]. Lastly, ARRF requires a minimum of two round-trips to reassemble the full response.

2) QBF [33]. Unlike ARRF, this scheme uses only standard record Type(s) and fragments *raw* signature/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. Fragments are requested by encoding the desired fragment number in the `QNAME` field of the Question section. Additionally, QBF is secure against memory-depletion attacks and can reconstruct a DNS response in one round-trip.

1.2 Benefits of TCP Fallback over Fragmentation Schemes

In this subsection, we contrast the two foregoing transport methods: Standard DNS with TCP fallback and UDP-based fragmentation schemes (ARRF/QBF).

Flow/Congestion Control. Both ARRF (in 2nd round-trip) and QBF (in 1st round-trip) send multiple DNS over UDP packets in parallel. Considering the absence of a TCP-like flow control in UDP, this deluge of DNS packets can potentially exhaust the bandwidth of busy resolvers and nameservers (which typically handle thousands of queries per second). Moreover, UDP's lack of a TCP-like congestion control could conceivably overwhelm the intermediate routers (leading to packet drops) or middleboxes such as load balancers, firewalls, etc.

Packet Loss. Unlike TCP, UDP does not guarantee a reliable delivery of packets. In ARRF/QBF, as the number of signatures to transmit or the sizes thereof grow (from configuring higher NIST levels), the number of DNS packets that need to be exchanged also naturally increases. 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 provide a perspective, given a 1% packet 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$^+$-128s session will require an extra round-trip.

DDoS Amplification and Reflection. Another concern with ARRF/QBF is their potential to be exploited for a DDoS attack [23,34,38], wherein small DNS queries with a spoofed source IP address cause large DNS responses (amplification) to be sent out from a nameserver to the target IP device (reflection), eventually overwhelming the latter or the network thereof. In one of the major DDoS events, attackers were able to generate 300 Gbps of traffic on a Tier 1 provider using open DNS resolvers [1].

Note that conducting such off-path attacks over TCP is not feasible because of the 3-way TCP handshake. This is because the client's query is forwarded to the DNS software only after receiving a valid[2] client `ACK` to the server `SYN`.

Performance. Because of the three-way handshake and the memory required to maintain the Transmission Control Block (TCB), TCP is generally slower and more resource intensive than UDP (which is just a thin wrapper around an IP packet for providing port numbers). This performance penalty is exacerbated in Standard DNS since a TCP fallback only occurs after the initial UDP response is explicitly marked `TC` by the nameserver.

Note that directly initiating a TCP connection (bypassing the UDP flow) is suboptimal because it is not straightforward to predict whether a response would be marked `TC` by the nameserver. Truncation depends on several factors (often unknown to the resolver) such as: the algorithm[3] with which the zone is signed, nameserver's `max-udp-size`, the number of resource records returned in each RRSet, whether `minimal-responses` are on/off, whether the response is `NXDOMAIN` containing `NSEC(3)` data, etc.

An alternative that can improve TCP performance in DNS is TCP Fast Open (TFO) [14]. In the initial `SYN` and `SYN-ACK` phase, a client obtains a TFO cookie (using a new TCP `option`) from a server. In later connections, the client supplies this cookie along with the request data in the `SYN` packet itself. The server, on verifying the cookie, makes the client's request available to the application, and subsequently starts streaming response packets while the handshake is still in progress. Unfortunately, a non-trivial number of middleboxes drop packets with unknown TCP options or `SYN` packets with data [20,29,25].

1.3 Our Contributions

In this work, we bridge the performance gap between Standard DNS and fragmentation schemes. We propose TurboDNS: a backward-compatible protocol that eliminates *two* rounds trips from the Standard DNS flow. In particular, the resolver sends a `SYN`, along with a cryptographic cookie, inside the initial DNS over UDP query. Upon truncation, the nameserver also includes a corresponding `SYN-ACK` in the `TC` response. Additionally, on a successful validation of the cookie, the nameserver forwards the resolver's query to the DNS software and immediately begins streaming the full DNS response over TCP.

A comparison of TurboDNS with other transport methods is depicted in Table 2. We now outline the salient benefits of TurboDNS.

Fast 1-RTT Resolution. DNSSEC over TurboDNS, with either Falcon-512 or Dilithium-2 as the zone algorithm, is $2\times$ as fast as Standard DNS (SD) in resolving QTYPE `A` queries. Moreover TurboDNS, being 30% faster than ARRF, matches the resolution speeds of QBF. Refer Fig. 2.

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

[3] To be specific, the algorithm of the Zone Signing Key (ZSK). The algorithm of the Key Signing Key (KSK) can be inferred from the DS record received from the zone's parent during the referral. Consult §2.2 for a primer on ZSK, KSK and DS record.

Table 2. A comparison of TurboDNS with Std. DNS (TCP fallback) and ARRF/QBF

	TurboDNS	**Standard DNS (SD)**	**ARRF/QBF**
Fast resolution	✓	✗	✓
No packet flooding	✓	✓	✗
Reliable	✓	✓	✗
DDoS resistant	✓	✓	✗

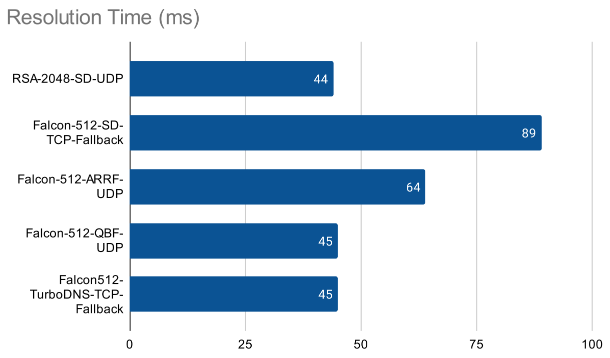

Fig. 2. A comparison of DNSSEC resolution times

TCP Robustness. TurboDNS suffers from none of the issues that affect UDP-based fragmentation schemes. With TCP's flow and congestion control, the end-points along with the intermediate devices are not overwhelmed with packets. Secondly, thanks to TCP's re-transmission mechanism, any dropped segment is recovered, thereby ensuring reliability. Finally, owing to cookie validation, TurboDNS cannot be misused as a DDoS amplifier and reflector.

Backward Compatibility. TurboDNS uses only RFC standardized resource record Type(s) and wire format to ensure that messages pass through even the most stringent of firewalls. Moreover, TurboDNS gracefully falls back to the Standard DNS flow should one of the end-points happen to be protocol-oblivious.

We implement TurboDNS as a daemon that runs atop the DNS software (such as BIND or PowerDNS) of resolvers/nameservers. With the daemon in place, no changes to the TCP kernel stack or the DNS software are required. The daemon also has the ability of re-computing the TCP checksum, maintaining packet ordering and dropping superfluous re-transmit packets.

Availability. The software artifact germane to this work is available at: https://github.com/aditya-asr/turbo-dns.

2 Preliminaries

Notations. The term *resource record* (RR) is often referred to as simply a *record.* || represents concatenation. X $\rightarrow$ Y denotes member Y of an abstract structure X. For *e.g.,* HEADER $\rightarrow$ `ID` denotes the `ID` field in the DNS HEADER. 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. (A)NS is short for (Authoritative) Name Server. While writing fully qualified domain names (FQDNs). we omit the root label (*i.e.* the trailing period (.) as in `example.com.`).

2.1 Domain Name System (DNS)

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

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 Type `A` record for `www.example.com`.

DNS Lookup. To retrieve the IP address of `www.example.com`, the client (or more precisely, the *stub resolver*) sends a *recursive* QTYPE `A` DNS query to its resolver (the 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 (`.`) ANS, which subsequently responds with the following records: 1) A Type `NS` record containing the domain name of `com` ANS 2) A Type `A` record containing the IP of `com` ANS.
2. It sends a QTYPE `NS` query to the `com` ANS, which responds with the following records: 1) A Type `NS` record containing the domain of `example.com` ANS 2) A Type `A` record containing the IP of `example.com` ANS.
3. It sends a QTYPE `A` query to the `example.com` ANS, which finally responds with a Type `A` record containing the IP of `www.example.com`.
4. It caches and forwards the received IP to the client.

Wire Format. A DNS message is divided into five sections: HEADER, Question, Answer, Authority, and Additional. 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 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` for Internet). The last three sections (Answer, Authority, and Additional) have the same format: a possibly empty list of concatenated DNS records.

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)
— `AD`: whether response has authenticated data (1) or not (0)
— `RCode`: (0) - no error; (1) or `FORMERR` - query was malformed; (3) or `NXDOMAIN` - domain name does not exist

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), `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 [40] introduces a pseudo-record called OPT (short for *options*) in the Additional section of a DNS message. Unlike traditional 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 or UDP payload size) in OPT $\rightarrow$ `CLASS`. The requester also indicates its ability to handle DNSSEC records by setting the `DO` (DNSSEC OK) bit in OPT $\rightarrow$ `TTL`. OPT $\rightarrow$ `RDATA` contains a set of options, with each option encoded as shown in Table 4.

`Option-Code 10` is assigned to DNS cookies [17] which provide protection against common off-path attacks such as denial-of-service, cache poisoning, and answer forgery. The client cookie and the server cookie are calculated as:

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

Table 4. OPT `RDATA` Wire Format

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`Option-Code`															
`Option-Length`															
`Option-Data` ...															

2.2 DNS Security Extensions (DNSSEC)

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

1) RRSIG. A signature is computed using a secret key (discussed below) over a set (called an RRset) of DNS records that have the same `NAME`, `CLASS` and `TYPE`. The resulting signature is stored in an RRSIG record (consult Table 5).

Table 5. 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 RRs signed			
`Signature Expiration`	When the signature expires			
`Signature Inception`	When the records were signed			
`Key Tag`	ID of the key that can verify the signature			
`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			

2) DNSKEY. A DNSKEY record (refer Table 6) 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 everything else. When a resolver receives a DNS response with an RRSIG record, it uses the associated DNSKEY record to verify the signature contained therein.

3) Delegation Signer (DS). The DS record (see Table 7) plays a vital role in establishing a secure chain of trust between parent and child zones.

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

Table 6. DNSKEY Wire Format

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

Whenever a resolver verifies RRSIGs using the $\mathsf{ZSK}_{\mathsf{pk}}$ of a child, it must also ascertain the authenticity of that key. Recall that the DNSKEY RRset containing $\mathsf{ZSK}_{\mathsf{pk}}$ and $\mathsf{KSK}_{\mathsf{pk}}$ is signed using the child's $\mathsf{KSK}_{\mathsf{sk}}$. Since KSK is ultimately self-signed, a resolver must also connect the trust thereof with the child's parent.

To aid resolvers in this endeavour, the child generates a hash of its $\mathsf{KSK}_{\mathsf{pk}}$ and shares it with its parent in a DS record. During a DNS lookup, when a resolver is referred to a child by its parent, the latter provides a DS record containing the hash of the child's $\mathsf{KSK}_{\mathsf{pk}}$. This DS record is what indicates to the resolver that the child zone is DNSSEC-enabled. More importantly, the parent also furnishes an RRSIG on this DS record using its own $\mathsf{ZSK}_{\mathsf{sk}}$. To validate the child zone's $\mathsf{KSK}_{\mathsf{pk}}$, 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 $\mathsf{ZSK}_{\mathsf{pk}}$ of the parent.

Table 7. 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	$\mathsf{hash}(\mathsf{DNSKEY} \rightarrow \texttt{NAME} \parallel \mathsf{DNSKEY} \rightarrow \texttt{RDATA})$			

DNSSEC Lookup. This is similar to the DNS lookup described in §2.1, except that the resolver now sets the `DO` bit in its 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 $\mathsf{ZSK}_{\mathsf{sk}}$. Additionally, it sends (on an explicit QTYPE `DNSKEY` query) (.)'s DNSKEYs and RRSIG thereon created with (.)'s $\mathsf{KSK}_{\mathsf{sk}}$. Here, we assume the resolver already holds (.)'s $\mathsf{KSK}_{\mathsf{pk}}$ as the *trust anchor*.
2. The `com` nameserver also sends `example.com`'s DS and RRSIG thereon created with `com`'s $\mathsf{ZSK}_{\mathsf{sk}}$. Additionally, it sends (on an explicit QTYPE `DNSKEY` query) `com`'s DNSKEYs and RRSIG thereon created with `com`'s $\mathsf{KSK}_{\mathsf{sk}}$.

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

On a successful DNSSEC validation, the resolver sends its answer response to the client with HEADER $\rightarrow$ `AD` set.

2.3 TCP Segment

Table 8 illustrates the TCP segment wire format. The TCP Header (includes all fields of the segment except `Data`) ranges from 20−60 bytes in size. The `Checksum` is calculated over: TCP *Pseudo-Header* || TCP Header (with `Checksum` = 0) || TCP `Data`. Note that a correct `Checksum` is mandatory for a TCP segment. Lastly, the `Control Bits` in the Header consists of six bits as shown in Table 9.

Table 8. TCP Segment Wire Format

0 1 2 3	4 5 6 7 8 9	10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
`Source Port`			`Destination Port`
`Sequence Number`			
`Acknowledgment Number`			
`Offset`	`Reserved`	`Control Bits`	`Window`
`Checksum`			`Urgent Pointer`
`TCP Options + Padding` ...			
`Data`			

Table 9. TCP Segment `Control Bits`

0	1	2	3	4	5
`URG`	`ACK`	`PSH`	`RST`	`SYN`	`FIN`

— `ACK`: Segment is carrying an *acknowledgment*
— `PSH`: *Push* data immediately to the receiver's application
— `SYN`: *Synchronize* sequence numbers
— `FIN`: *Finish* (close) the connection

3 TurboDNS

TurboDNS is backward-compatible protocol that eliminates two round-trips from Standard DNS with TCP fallback (refer Fig. 1). At a high level, it works as follows: the initial DNS/UDP query from the client carries a `SYN`. On the server side, if the resulting response exceeds the UDP limits, the `SYN` is consumed and a `SYN-ACK` is included in the `TC` response.

If the client and the server are interacting for the first time, the latter computes a cookie on the client's IP address and includes it in the DNS response. In subsequent interactions, the client presents the cookie in its initial DNS/UDP query. On a successful verification of the cookie, the server starts streaming the full DNS/TCP response after sending the `SYN-ACK` in the `TC` response.

TurboDNS works via a daemon that runs *atop* the DNS stack of resolvers and nameservers: the implication being that the DNS software remains completely unaware of the daemon. Thus, with the daemon installed, no modifications to the DNS software or the TCP stack are required.

TD-Cookie. Similar to TFO [14], at the core of TurboDNS is a cryptographic cookie (hereafter referred to as `TD-Cookie` to differentiate it from regular DNS cookies[5]) that verifies a client's IP ownership without requiring the server to maintain a *per-client* state. The `TD-Cookie` prevents an adversary from 1) exhausting the server's resources by flooding DNS queries with spoofed `SYN`s or 2) mounting an amplified reflection attack on random victims.

When a client and a server interact for the first time (either over UDP or TCP), the server daemon generates a `TD-Cookie` as HMAC-SHA256-8(sk, Client IP), where sk is a secret key held by the daemon and 8 denotes the HMAC output truncated to first 8 bytes (64-bit entropy). The `TD-Cookie` is then sent to the client in OPT $\rightarrow$ `RDATA` under the experimental `Option-Code = 65001`.

Note that TurboDNS does not use regular DNS cookies to authenticate clients. Since the daemon runs on top of the DNS software, it does not have access to the `Server Secret` that is being used to compute the DNS Server Cookie.

We now elucidate the TurboDNS protocol with the aid of Fig. 3 and Fig. 4. Consider a resolver and an `example` authoritative nameserver (ANS), each with TurboDNS daemon installed. Further presume that the resolver daemon has already obtained a valid `TD-Cookie` from a prior interaction with the ANS.

3.1 Client-side: Sending Query

Assume that the resolver (here, BIND) sends a QTYPE `A` query (say, Q) to the ANS asking the IPv4 address of `test.example`. The resolver daemon, on intercepting the outbound Q, *simulates*[6] a `TC` response from the ANS, thereby triggering a TCP fallback on BIND. The latter subsequently attempts a TCP handshake by sending a `SYN` packet.

[5] Refer the OPT record subsection in §2.1 for a primer on regular DNS cookies.
[6] Includes swapping source and destination IP/port.

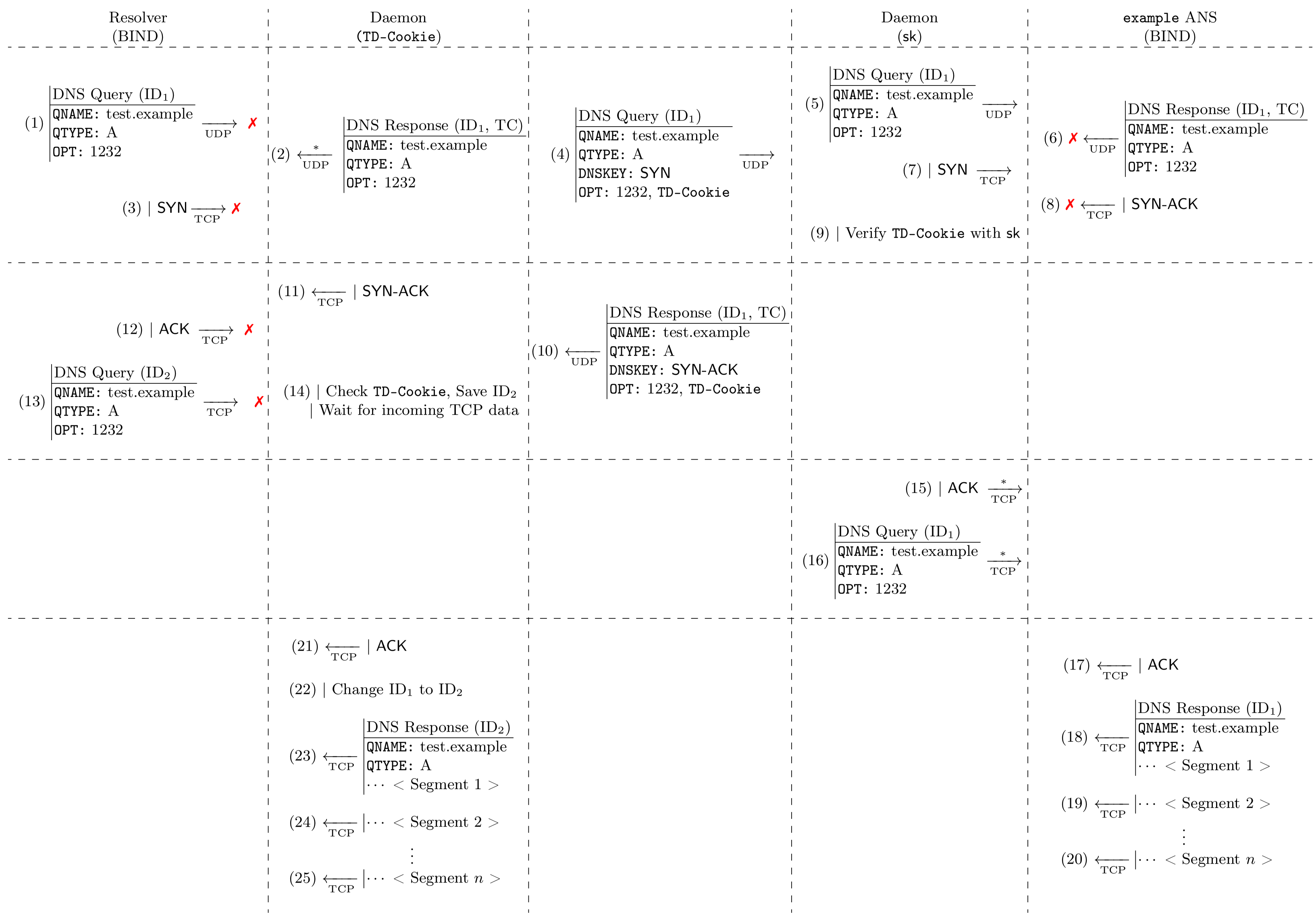
Resolver (BIND)
Daemon (TD-Cookie)
Daemon (sk)
example ANS (BIND)
(1) DNS Query (ID1) QNAME: test.example QTYPE: A OPT: 1232 UDP ✗
(2) * UDP DNS Response (ID1, TC) QNAME: test.example QTYPE: A OPT: 1232
(3) | SYN TCP ✗
(4) DNS Query (ID1) QNAME: test.example QTYPE: A DNSKEY: SYN OPT: 1232, TD-Cookie UDP
(5) DNS Query (ID1) QNAME: test.example QTYPE: A OPT: 1232 UDP
(6) ✗ UDP DNS Response (ID1, TC) QNAME: test.example QTYPE: A OPT: 1232
(7) | SYN TCP
(8) ✗ TCP | SYN-ACK
(9) | Verify TD-Cookie with sk
(10) UDP DNS Response (ID1, TC) QNAME: test.example QTYPE: A DNSKEY: SYN-ACK OPT: 1232, TD-Cookie
(11) TCP | SYN-ACK
(12) | ACK TCP ✗
(13) DNS Query (ID2) QNAME: test.example QTYPE: A OPT: 1232 TCP ✗
(14) | Check TD-Cookie, Save ID2 | Wait for incoming TCP data
(15) | ACK * TCP
(16) DNS Query (ID1) QNAME: test.example QTYPE: A OPT: 1232 * TCP
(17) TCP | ACK
(18) TCP DNS Response (ID1) QNAME: test.example QTYPE: A ··· < Segment 1 >
(19) TCP |··· < Segment 2 >
(20) TCP |··· < Segment n >
(21) TCP | ACK
(22) | Change ID1 to ID2
(23) TCP DNS Response (ID2) QNAME: test.example QTYPE: A ··· < Segment 1 >
(24) TCP |··· < Segment 2 >
(25) TCP |··· < Segment n >

Fig. 3. An overview of TurboDNS when response is marked truncated (TC)

* packet is simulated by the daemon
✗ packet is dropped by the daemon

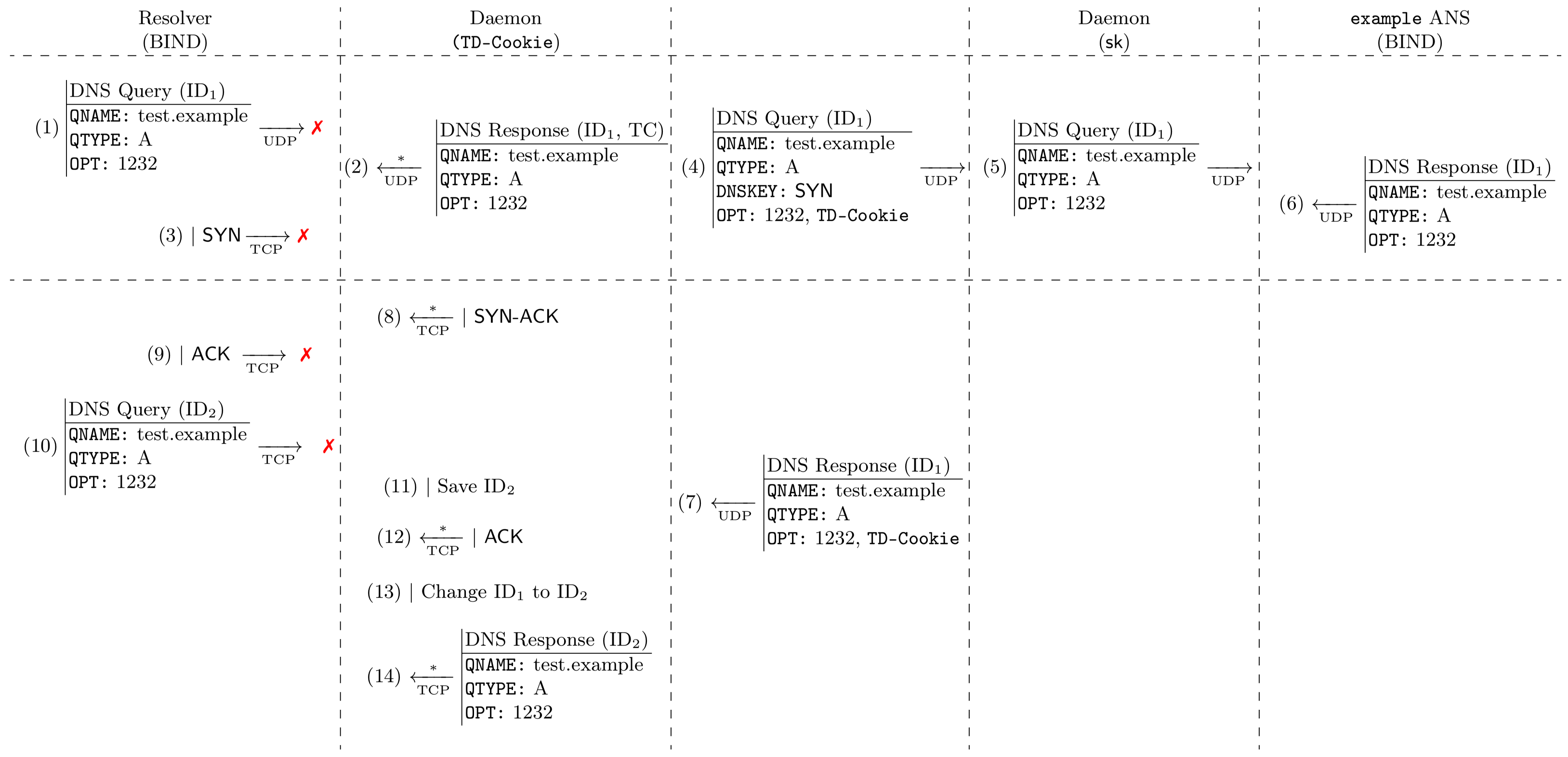
Resolver (BIND)
Daemon (TD-Cookie)
Daemon (sk)
example ANS (BIND)
(1) DNS Query (ID1) QNAME: test.example QTYPE: A OPT: 1232 UDP ✗
(2) * UDP DNS Response (ID1, TC) QNAME: test.example QTYPE: A OPT: 1232
(3) | SYN TCP ✗
(4) DNS Query (ID1) QNAME: test.example QTYPE: A DNSKEY: SYN OPT: 1232, TD-Cookie UDP
(5) DNS Query (ID1) QNAME: test.example QTYPE: A OPT: 1232 UDP
(6) UDP DNS Response (ID1) QNAME: test.example QTYPE: A OPT: 1232
(7) UDP DNS Response (ID1) QNAME: test.example QTYPE: A OPT: 1232, TD-Cookie
(8) * TCP | SYN-ACK
(9) | ACK TCP ✗
(10) DNS Query (ID2) QNAME: test.example QTYPE: A OPT: 1232 TCP ✗
(11) | Save ID2
(12) * TCP | ACK
(13) | Change ID1 to ID2
(14) * TCP DNS Response (ID2) QNAME: test.example QTYPE: A OPT: 1232

Fig. 4. An overview of TurboDNS when response is not marked truncated (TC)

* packet is simulated by the daemon
✗ packet is dropped by the daemon

The daemon intercepts the `SYN`, encapsulates it inside a DNSKEY record, and then inserts the record in Q's Additional section. We choose a DNSKEY record for encapsulation since it has minimal `RDATA` fields, with the `Public Key` field being opaque (*i.e.* its content is not meaningful to a protocol-unaware entity). Lastly, the daemon inserts the `TD-Cookie` in the OPT record and sends Q.

Concisely, the steps performed by the resolver daemon, on intercepting the outgoing query Q/UDP, are as follows:

1. Create a copy (say, R') of Q.
2. $\boxed{\text{Forcing TCP Fallback}}$
   (a) Modify R' as follows[7]:
      — Set $R' \rightarrow$ HEADER $\rightarrow$ `QR` $= 1$
      — Set $R' \rightarrow$ HEADER $\rightarrow$ `AA` $= 1$
      — Set $R' \rightarrow$ HEADER $\rightarrow$ `TC` $= 1$
   (b) Send R'/UDP back to BIND.
   (c) Receive `SYN`/TCP from BIND.
3. $\boxed{\text{Embedding SYN in Q}}$
   (a) Create a generic DNSKEY record.
      — Set DNSKEY $\rightarrow$ `Algorithm` $=$ `TurboDNS`
      — Set DNSKEY $\rightarrow$ `Public Key` $=$ `SYN`[8]
   (b) Insert DNSKEY in Q $\rightarrow$ Additional section.
4. $\boxed{\text{Embedding TD-Cookie in Q}}$
   (a) Insert `TD-Cookie` in Q $\rightarrow$ OPT $\rightarrow$ `RDATA`.
      — Set `Option-Code` $=$ `65501`
      — Set `Option-Length` $=$ `8`
      — Set `Option-Data` $=$ `TD-Cookie`
5. Send Q/UDP to ANS.

3.2 Server-side: Sending Response

The ANS daemon, on detecting the incoming query Q, forwards[9] it to BIND. The resulting response (say, R) will have either of the following status:

1. R is not marked `TC`. In this case, the daemon forwards the response *as it is* to the resolver.

2. R is marked `TC`. In this event, the daemon extracts the `SYN` from Q and sends it to BIND. The subsequent `SYN-ACK` from BIND is intercepted and encapsulated in a DNSKEY record, which is then inserted in R's Additional section.

Thereafter, the daemon computes HMAC-256-8(sk, Q's source IP) and compares it against the provided `TD-Cookie` in the OPT record of Q. If the values match, the daemon inserts the same `TD-Cookie` in the OPT record of R. Otherwise, it inserts the newly computed cookie. R is then sent to the resolver.

[7]For a primer on the various HEADER flags, refer Table 3.
[8]The whole IP packet.
[9]Removing DNSKEY and `TD-Cookie` is optional as they would be ignored by BIND.

After verifying `TD-Cookie`, the nameserver daemon *simulates* the resolver by sending an `ACK` along with Q to BIND over TCP. The ensuing TCP responses are then forwarded to the resolver.

Succinctly, the nameserver daemon performs the following operations after intercepting the incoming query Q/UDP:

1. Forward Q to BIND and get response R/UDP.
2. If R is not `TC`, forward it to resolver. Else, proceed as below.
3. $\boxed{\text{Simulating TCP Handshake}}$
   (a) Extract `SYN` from Q.
   (b) Send `SYN`/TCP to BIND and get `SYN-ACK`.
4. $\boxed{\text{Embedding SYN-ACK in R}}$
   (a) Create a generic DNSKEY record.
   — Set DNSKEY $\rightarrow$ `Algorithm` = `TurboDNS`
   — Set DNSKEY $\rightarrow$ `Public Key` = `SYN-ACK`
   (b) Insert DNSKEY in R $\rightarrow$ Additional section.
5. $\boxed{\text{Verifying TD-Cookie and Embedding TD-Cookie/HMAC in R}}$
   (a) Extract `TD-Cookie` from Q.
   (b) Verify HMAC(sk, Q's src IP) $\overset{?}{=}$ `TD-Cookie`
   — If success, insert `TD-Cookie` in R $\rightarrow$ OPT $\rightarrow$ `RDATA`.
   — If failure, insert HMAC output in R $\rightarrow$ OPT $\rightarrow$ `RDATA`.
6. Send R/UDP to resolver.
7. $\boxed{\text{Simulating DNS/TCP Query}}$
   (a) If `TD-Cookie` was verified, then
   i. Simulate resolver and send (`ACK`+Q)/TCP to BIND.
   ii. Forward BIND TCP responses to resolver.

3.3 Client-side: Receiving Response

The daemon on the resolver, on intercepting R, checks the status thereof and proceeds accordingly as follows:

R is not marked TC. In this scenario, the daemon *simulates* the nameserver by sending a `SYN-ACK` to BIND, which subsequently sends over TCP: 1) an `ACK` and 2) Q with a new HEADER $\rightarrow$ `ID`. Now, the daemon simply changes[10] the `ID` in R to the new value and sends it to BIND over TCP.

R is marked TC. Here, the daemon extracts the `SYN-ACK` from R and sends it to BIND, the latter subsequently sending an `ACK` and Q with a new `ID`.

The daemon then checks the sent TurboDNS cookie in R to infer whether the validation of `TD-Cookie` succeeded on the ANS or not (*i.e.* whether the server has echoed back `TD-Cookie` or not). If negative, from this point onwards, the daemon forwards the outgoing/incoming traffic *as it is*. If positive, the daemon drops both the packets (`ACK` and Q), while saving the new `ID` in its state.

[10]Includes re-computing the TCP `Checksum`.

In due course, when TCP response packets arrive from the ANS after a successful cookie validation, the daemon intercepts the first data segment[11] and updates[10] `ID` therein with the new value, before forwarding it to BIND. The rest of the trailing segments are forwarded *as it is* to BIND.

In short, the daemon, on intercepting R/UDP, proceeds as follows:

— If R is not marked `TC`, simulate ANS as below.
  1. $\boxed{\text{Simulating TCP Handshake}}$
     (a) Send `SYN-ACK`/TCP to BIND.
  2. Intercept (`ACK`+Q)/TCP with new Q $\rightarrow$ HEADER $\rightarrow$ `ID`.
  3. $\boxed{\text{Simulating DNS/TCP Response}}$
     (a) Set R $\rightarrow$ HEADER $\rightarrow$ `ID` = Q $\rightarrow$ HEADER $\rightarrow$ `ID`.
     (b) Send R/TCP to BIND.

— If R is marked `TC`, proceed as below[12].
  1. $\boxed{\text{Simulating TCP Handshake}}$
     (a) Extract `SYN-ACK` from R.
     (b) Send `SYN-ACK`/TCP to BIND.
  2. Intercept (`ACK`+Q)/TCP with new Q $\rightarrow$ HEADER $\rightarrow$ `ID`.
  3. Intercept 1st TCP data segment from ANS.
  4. Update DNS HEADER $\rightarrow$ `ID` in 1st segment with Q $\rightarrow$ HEADER $\rightarrow$ `ID`.
  5. $\boxed{\text{Simulating DNS/TCP Response}}$
     (a) Send 1st segment to BIND.
  6. Forward all trailing segments to BIND.

3.4 Backward Compatibility

We now discuss what happens in case only one of the end-points implements TurboDNS while the other one does not.

— TurboDNS-aware Resolver | TurboDNS-oblivious Server. In such a setting, the server will never send a `TD-Cookie` to the resolver. Furthermore, in resolver's query, the DNSKEY record containing `SYN` would be ignored by the server since the record is present in the Additional section. Thus, the flow will inevitably become that of Standard DNS.

— TurboDNS-oblivious Resolver | TurboDNS-aware Server. Here, the server will include a `TD-Cookie` in OPT $\rightarrow$ `RDATA` in its response to the resolver. However, any `Option-Code` not understood by the resolver would be ignored ([40], p. 8). Additionally, the server will not find a DNSKEY record containing `SYN` in the Additional section of resolver's query. Hence, the flow effectively reverts to that of Standard DNS.

[11] With a relative Server `Sequence Number` = 1

[12] Assuming `TD-Cookie` was accepted by the server daemon.

3.5 Other Remarks

Packet Ordering. In case of a `TC` response and a valid cookie, the `SYN-ACK` over DNS/UDP must arrive at the client before the trailing TCP data stream. To this effect, after sending the `SYN-ACK`, the server daemon waits an extra 1 ms before simulating the client `ACK`. However, depending upon network conditions, larger wait times may be necessary to maintain the packet ordering.
Segment Re-transmits. When the server's TCP stack receives the simulated client `ACK` within a few ms after `SYN-ACK`, its view as to the round-trip latency of the connection gets temporarily skewed (*i.e.* the actual RTT is generally larger). Thus, after the full DNS/TCP response has been streamed to the client, the server-side may reach the TCP Retransmission Timeout (RTO) before client `ACK`s arrive, leading to unwarranted segment re-transmits.

To avoid unnecessary bandwidth usage, the server daemon drops the packets in the *first* re-transmission attempt of BIND'S TCP, with the consequence that the latter then exponentially backs-off its RTO. In most cases, client `ACK`s will arrive before the expiration of the re-adjusted RTO.

4 Security Considerations

It is important to note that TurboDNS does not introduce any changes to the underlying cryptography of DNSSEC. Thus, the security guarantees of the DNSSEC protocol (*i.e.* origin authentication and integrity check of DNS resource records) remain completely intact.

On a high level, TurboDNS tweaks the Standard DNS flow by 1) Carrying TCP `SYN` and `SYN-ACK` packets inside DNS/UDP messages 2) Streaming TCP response data after authenticating the client with `TD-Cookie`. While the former tweak only changes the delivery mechanism of packets and thus does not introduce any security vulnerability, the latter, however, requires some additional security considerations against *on-path* and *off-path* attackers.

Just like regular DNS cookies, `TD-Cookie` does not protect against an on-path adversary (*i.e.* an attacker who can observe the plaintext DNS traffic such as an on-path router, bridge, or any device on an on-path shared link). An on-path attacker who learns the `TD-Cookie` generated for a client can later send spoofed DNS requests on behalf of that client. We discuss the appropriate counter-measures such as response rate limiting (RLL) and secret key rollover later in this section. An off-path attack, however, is infeasible since it requires the attacker to simultaneously guess the `TD-Cookie` (along with the regular DNS cookie) for a particular client IP address. Security can be further strengthened by increasing the `TD-Cookie` size.

Broadly, there are two kinds of attacks that can be performed by an attacker.

1. **Resource Exhaustion with Stolen `TD-Cookies`.** If an attacker manages to steal `TD-Cookie`s from compromised clients, it can then flood DNS queries containing spoofed `SYN`s and valid cookies, thereby potentially exhausting a server's CPU and memory resources.

In this scenario, the best defence is to rate-limit the number of TurboDNS requests that will be processed per `TD-Cookie` (and consequently, per source IP address) in a given time frame. Additionally, the server daemon should also rotate its `sk` on a frequent basis.

2. **Amplified Reflection Attacks using NATs.** Since multiple clients behind a NAT box share the same external IP address, a `TD-Cookie` issued by a server daemon to any client will also be valid for the rest of the NATed clients. Consider an attacker behind a NAT that first obtains a valid `TD-Cookie` from server-side. Thereafter, it uses the cookie to issue a flood of DNS queries with spoofed `SYN`s of a victim client (behind the same NAT). Since cookie validation will succeed on the server daemon, large DNSSEC traffic (amplification) would be sent to the victim (reflection), potentially overwhelming the latter. Fortunately, a simple tweak obviates this vulnerability. The `TD-Cookie` can be calculated as HMAC-SHA256-8(`sk`, Client IP || Client Cookie), where Client Cookie is the regular DNS cookie. Since a unique Client Secret is mixed in Client Cookie calculation, each client computes a unique Client Cookie, thus resulting in different `TD-Cookie`s being issued to NATed clients.

5 Evaluation

5.1 Implementation

To assess the TurboDNS's performance, we develop a daemon that runs on top of a DNS software (such as BIND 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 or TCP stack.

Software Setup. We use the source code of QBF [33] as base to build the TurboDNS daemon. The DNS software is a BIND 9.19.17 fork [2] which supports NIST level I PQC signatures. In the fork, we further add support for Level V Falcon and Dilithium schemes. The cryptographic stack is `openssl` 3.2, `liboqs` 0.10.0 [43] and `oqs-provider` 0.6.0. The daemon is written in C and uses `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`. 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 (the user) 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 an Ubuntu 22.04 Docker container with experiment-specific bandwidth and latency constraints. The TurboDNS daemon is installed on the resolver and the ANS. The EDNS0 buffer 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 zone file of 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* when generating responses for iterative queries. Such a response is called *non-minimal* and represents the worst-case scenario in terms of message size. Refer Table 10 for the number and the type of resource records contained in a *non-minimal* QTYPE `A` response from the ANS. To facilitate modifications to DNS messages without re-adjusting compression name pointers, we also set `message-compression no;` in `named.conf`.

Table 10. A non-minimal QTYPE A response

Header Section
Question Section
QNAME = `test.example`
QTYPE = `A`
QCLASS = `IN`
Answer Section
Type `A` RR
RRSIG
Authority Section
Type `NS` RR
RRSIG
Additional Section
Type `A` RR
RRSIG
OPT

- 1 Type `A` record in Answer section having the answer IP and 1 covering RRSIG
- 1 Type `NS` record in Authoritative section containing the nameserver's domain name and 1 covering RRSIG
- 1 Type `A` record in Additional section containing the nameserver's IP address and 1 covering RRSIG

5.2 Experiments and Results

We comprehensively assess TurboDNS's performance against Standard DNS and fragmentation schemes (ARRF and QBF) in terms of resolution times. We conduct two experiments targeting NIST level I and V, respectively.

Before the start of an experiment, the resolver pre-fetches the Type `DNSKEY` and `NS` records of all the zones. 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. Moreover, as a side effect of the pre-fetching, the resolver daemon obtains the `TD-Cookie` from the server-side.

We measure the DNS query resolution speed with each participant configured with the following networking capabilities[13,14]:

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 query resolution time. That is, the average time elapsed between the client sending its query and subsequently receiving a DNSSEC validated response (with HEADER $\rightarrow$ `AD` set) from the resolver.

a) Experiment 1. We target NIST level I parameters and sign the zone file with Falcon-512, Dilithium-2 and SPHINCS$^+$-128s. For a comparison with classical algorithms, we sign the zone file with RSA-2048 and ECDSA P-256.

Results and Discussion. The results of the experiment are tabulated in Table 11. Observe that DNSSEC over TurboDNS (TD), with either Falcon or Dilithium as the underlying scheme, has practically the same resolution time as that of the currently deployed ECDSA-SD and RSA-SD. Moreover, Falcon-TD and Dilithium-TD, being $\sim 2\times$ faster than their Standard DNS (SD) counterparts, also match the resolution speeds of the corresponding QBF instances.

Interestingly, with SPHINCS$^+$, both TurboDNS and Standard DNS incur an extra round-trip compared to Falcon and Dilithium. This is because the size of a SPHINCS$^+$ QTYPE `A` response exceeds the `initcwnd` (initial congestion window) of 10 segments set in the TCP slow start algorithm [10,15]. Given the default MSS (Maximum Segment Size) of 1220 bytes, the size of `initcwnd` is $10 \times 1220 = 12.2$ KB. The repercussion of exceeding this `initcwnd` is that after sending ~ 12.2 KB of data, the server waits for the resolver to `ACK` (acknowledge) the received packets before continuing with the rest of the transmission.

b) Experiment 2. We now target NIST level V parameters and sign the zone file with Falcon-1024 and Dilithium-5 (Refer Table 1 for a size comparison between the two schemes). This is the highest security level on offer and is likely excessive for DNSSEC. We omit testing SPHINCS$^+$ since the resulting response would exceed 64 KB, the maximum possible size for a DNS message. We also exclude ARRF from this experiment as its performance can be extrapolated from that of QBF (*i.e.* ARRF is a round-trip slower than QBF).

Results and Discussion. The results of the experiment are shown in Table 11. Compared to Experiment 1, while Falcon-TD/SD remain consistent with their resolution speeds, Dilithium-TD/SD suffer a penalty of an extra round-trip because of exceeding TCP's `initcwnd`.

[13]Latency being one-way, RTT = $2\times$ Latency.

[14]Packet loss rate is set to 0%. This is the best-case scenario for UDP-based fragmentation schemes in terms of performance.

Table 11. Average query resolution time (ms). SD : Standard DNS. TD : TurboDNS. TCP* : TCP Fallback. A: 100 Mbps, 10 ms. B: 1 Mbps, 100 ms.

Method	**Via**	**Time A** ± 1 **ms**	**Time B** ± 2 **ms**
ECDSA-P256-SD	UDP	44	407
RSA-2048-SD	UDP	44	408
Falcon-512-SD	TCP*	89	811
Dilithium-2-SD	TCP*	89	817
SPHINCS$^+$-128s-SD	TCP*	111	1025
Falcon-1024-SD	TCP*	89	812
Dilithium-5-SD	TCP*	111	1014
Falcon-512-ARRF	UDP	64	609
Dilithium-2-ARRF	UDP	65	613
SPHINCS$^+$-128s-ARRF	UDP	67	633
Falcon-512-QBF	UDP	45	410
Dilithium-2-QBF	UDP	46	415
SPHINCS$^+$-128s-QBF	UDP	48	436
Falcon-1024-QBF	UDP	45	411
Dilithium-5-QBF	UDP	47	426
Falcon-512-TD	TCP*	45	409
Dilithium-2-TD	TCP*	46	415
SPHINCS$^+$-128s-TD	TCP*	68	622
Falcon-1024-TD	TCP*	45	412
Dilithium-5-TD	TCP*	67	610

6 Related Work

Apart from ARRF [18] and QBF [33], earlier DNS-layer proposals included Sivaraman's draft [41] and ATR [42]. The former fragmented a large DNS response across multiple UDP datagrams, transmitting each fragment sequentially. The latter involved an ATR 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 because they sent multiple responses to one request. Many firewalls expect a single response packet per query. Furthermore, many resolvers immediately close their sockets after receiving the first response packet. Thus, there were concerns about ICMP flooding since for each trailing packet that could not be delivered, a *destination unreachable* packet would be sent back to the nameserver.

While both TFO [14] and TurboDNS employ a cookie-based mechanism to authenticate the client, the similarities end here. TFO requires support at the OS and application level, introduces a new TCP `option` for transporting the cookie, and involves `SYN` segments carrying data, the latter two leading to potential backward-incompatibility. On the other hand, TurboDNS uses the stock TCP kernel implementation and does not require any additional support from the DNS software. Moreover, TurboDNS's cookies are conveyed inside DNS messages in a backward-compatible manner.

In the domain of TLS, TurboTLS [6] establishes TLS connections in one less round-trip by sending the initial TLS handshake data over UDP rather than TCP (compare this with TurboDNS, which sends the TCP handshake data (`SYN` and `SYN-ACK`) inside DNS messages over UDP). Concurrently, a three-way TCP handshake is also carried out. Once the TCP connection is established, the client and the server complete the final flight of the TLS handshake over TCP and continue using it for transferring application data.

7 Conclusion

In this paper, we presented TurboDNS: a backward-compatible protocol that eliminates two round-trips from Standard DNS with TCP fallback. The TCP handshake data is exchanged in the initial DNS over UDP flight while the full DNS response is immediately streamed over TCP after authenticating the client with a cryptographic cookie. Our experiments show that DNSSEC over TurboDNS, with either Falcon-512 or Dilithium-2, is as fast as the presently deployed ECDSA P-256 and RSA-2048 in resolving `QTYPE A` queries.

References

1. The ddos that almost broke the internet. https://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet, accessed: 2024-07-09
2. Oqs-bind. https://github.com/Martyrshot/OQS-bind
3. Defragmenting dns - determining the optimal maximum udp response size for dns (2020), https://indico.dns-oarc.net/event/36/contributions/776/, accessed: 2024-07-09
4. Dns flag day. https://www.dnsflagday.net/2020/ (2020)
5. Aas, J., Barnes, R., Case, B., Durumeric, Z., Eckersley, P., Flores-López, A., Halderman, J.A., Hoffman-Andrews, J., Kasten, J., Rescorla, E., Schoen, S., Warren, B.: Let's encrypt: An automated certificate authority to encrypt the entire web. In: SIGSAC CCS (2019)
6. Aguilar-Melchor, C., Bailleux, T., Goertzen, J., Guinet, A., Joseph, D., Stebila, D.: Turbotls: Tls connection establishment with 1 less round trip (2024), https://arxiv.org/abs/2302.05311
7. Ariyapperuma, S., Mitchell, C.J.: Security vulnerabilities in dns and dnssec. In: ARES (2007)
8. Atkins, D., Austein, R.: Threat analysis of the domain name system (dns). RFC 3833 (2004)
9. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs+ signature framework. In: SIGSAC CCS (2019)
10. Blanton, E., Paxson, D.V., Allman, M.: Tcp congestion control. RFC 5681 (2009)
11. Bonica, R., Baker, F., Huston, G., Hinden, B., Trøan, O., Gont, F.: Ip fragmentation considered fragile. RFC 8900 (2020)
12. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehle, D.: Crystals - kyber: A cca-secure module-lattice-based kem. In: EuroS&P (2018)
13. Bush, R., Austein, R.: The Resource Public Key Infrastructure (RPKI) to Router Protocol, Version 1. RFC 8210 (2017)
14. Cheng, Y., Chu, J., Radhakrishnan, S., Jain, A.: TCP Fast Open. RFC 7413 (2014)
15. Chu, J., Dukkipati, N., Cheng, Y., Mathis, M.: Increasing tcp's initial window. RFC 6928 (2013)
16. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium: A lattice-based digital signature scheme. IACR TCHES (2018)
17. Eastlake, D.E., Andrews, M.P.: Domain name system (dns) cookies. RFC 7873 (2016)
18. Goertzen, J., Stebila, D.: Post-quantum signatures in DNSSEC via request-based fragmentation. In: PQCrypto (2023)
19. Herzberg, A., Shulman, H.: Fragmentation considered poisonous, or: One-domain-to-rule-them-all.org. In: IEEE CNS (2013)
20. Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., Tokuda, H.: Is it still possible to extend tcp? In: IMC (2011)
21. Jeitner, P., Shulman, H.: Injection attacks reloaded: Tunnelling malicious payloads over DNS. In: USENIX (2021)
22. Kaminsky, D.: Black ops: 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 (2008), accessed: 2024-07-09

23. Kampanakis, P., Lepoint, T.: Vision paper: Do we need to change some things? In: SSR (2023)
24. Kaufman, C., Perlman, R., Sommerfeld, B.: Dos protection for udp-based protocols. In: SIGSAC CCS (2003)
25. Langley, A.: Probing the viability of tcp extensions. https://www.imperialviolet.org/binary/ecntest.pdf, accessed: 2024-07-25
26. Lepinski, M., Kent, S.: An Infrastructure to Support Secure Internet Routing. RFC 6480 (2012)
27. Man, K., Qian, Z., Wang, Z., Zheng, X., Huang, Y., Duan, H.: Dns cache poisoning attack reloaded: Revolutions with side channels. In: SIGSAC CCS (2020)
28. Man, K., Zhou, X., Qian, Z.: Dns cache poisoning attack: Resurrections with side channels. In: SIGSAC CCS (2021)
29. Medina, A., Allman, M., Floyd, S.: Measuring interactions between transport protocols and middleboxes. In: IMC (2004)
30. Moura, G.C.M., Müller, M., Davids, M., Wullink, M., Hesselman, C.: Fragmentation, truncation, and timeouts: Are large dns messages falling to bits? In: PAM (2021)
31. Müller, M., de Jong, J., van Heesch, M., Overeinder, B., van Rijswijk-Deij, R.: Retrofitting post-quantum cryptography in internet protocols: a case study of dnssec. SIGCOMM CCR (2020)
32. Prest, T., Fouque, P., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon. tech. rep., national institute of standards and technology, available at. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022 (2022)
33. Rawat, A.S., Jhanwar, M.P.: Post-quantum dnssec over udp via qname-based fragmentation. In: SPACE (2023)
34. van Rijswijk-Deij, R., Sperotto, A., Pras, A.: Dnssec and its potential for ddos attacks: a comprehensive measurement study. In: IMC (2014)
35. Rose, S., Larson, M., Massey, D., Austein, R., Arends, R.: Dns security introduction and requirements. RFC 4033 (2005)
36. Rose, S., Larson, M., Massey, D., Austein, R., Arends, R.: Protocol modifications for the dns security extensions. RFC 4035 (2005)
37. Rose, S., Larson, M., Massey, D., Austein, R., Arends, R.: Resource records for the dns security extensions. RFC 4034 (2005)
38. Rossow, C.: Amplification hell: Revisiting network protocols for ddos abuse. In: NDSS (2014)
39. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SICOMP (1997)
40. da Silva Damas, J., Graff, M., Vixie, P.A.: Extension mechanisms for dns (edns(0)). RFC 6891 (2013)
41. Sivaraman, M., Kerr, S., Song, L.: Dns message fragments. https://datatracker.ietf.org/doc/draft-muks-dns-message-fragments/00/
42. Song, L., Wang, S.: Atr: Additional truncation response for large dns response. https://datatracker.ietf.org/doc/draft-song-atr-large-resp/03/
43. Stebila, D., Mosca, M.: Post-quantum key exchange for the internet and the open quantum safe project. In: SAC (2017)
44. Van Den Broek, G., Van Rijswijk-Deij, R., Sperotto, A., Pras, A.: Dnssec meets real world: dealing with unreachability caused by fragmentation. IEEE Communications Magazine (2014)