Providing DNS Security in Post-Quantum Era with Hash-based Signatures

Sevinj Jafarli

M.Sc. Thesis

Feb 2022

Supervisors:
prof. dr. ir. van Rijswijk - Deij, Roland
dr. ing. W. Hahn, Florian
dr. ir. Müller, Moritz

DACS Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

With the advent of quantum computers, current practices in DNSSEC will become vulnerable and obsolete since Shor's algorithm was proven to break public-key cryptography. Therefore, providing security of the DNS in the post-quantum era becomes the main challenge and point of interest for different research groups. In this thesis, we propose an innovative way of signing DNS using Merkle Tree and XMSS Hash-Based Signature Scheme as having been proven to be quantum-resistant. We suggest grouping resource records as leaves of a Merkle tree and signing the root of the tree with XMSS. In this scenario, the signature over the record is merely the intermediary hash nodes to recompute the root and the signature over 'DNSKEY' is the XMSS signed root node. Since the size of the tree determines the length of the authentication path (signature size) and hence what needs to be transmitted in DNS messages, larger trees will lead to an increase in the signature size as well as more time to update the tree and compute the signatures. Therefore, the objectives of this research are threefold. The first is to identify important variables from the proposed approach to be traded off. Second is to analyse and evaluate the impact of the variables on important DNS metrics such as *signature size*, *signing* and *verification speed*. Finally, checking the impact of the innovative grouping approach based on the popularity or update frequencies of the records on the *tree update frequencies* along with the metrics.

Results show that the identified variables have a major impact on the signing speed rather than verification. In the same zone, constructing smaller trees would yield more trees but less time to sign the entire zone than having a few large trees. In the case of many small trees, the number of roots to be signed increases and concatenating all the roots for further XMSS signing might lead to excessively large DNS messages. On the other hand, having large trees would mean fewer trees in the zone thus fewer tree roots. However, since a single tree contains more records, the likelihood of frequent tree updates is high which leads to the regeneration of the signature in the entire tree. We evaluated the impact of these trade-offs using a prototype and real-world data and show that it is possible to pick values for the three variables that lead to a feasible, implementable version of our proposed signing scheme. Consequently, a mix of small and large trees (variable size trees in the mid and large zones) are suggested as a common ground for the above trade-offs.

Contents

List of acronyms

DNS	Domain Name System
DNSSEC	Domain Name System Security Extensions
TLD	Top Level Domain
TTL	Time To Live
MTU	Maximum Transmission Unit
PKI	Public Key Infrastructure
RR	Resource Record
OTS	One Time Signature
MSS	Merkle Signature Scheme
XMSS	Extended Merkle Signature Scheme
XMSSMT	Extended Merkle Signature Scheme Multi Tree

Chapter 1

Introduction

The Domain Name System (DNS) provides the essence of accessing and reaching online resources on the internet. Similar to other internet protocols, securing DNS is an important responsibility of the operators. DNSSEC is the security extension for DNS that allows to verify the integrity and authenticity of the DNS messages. Nowadays, public-key cryptography offers security to various systems and network protocols. DNSSEC is one of them as the authentication and integrity of DNS data rely on the digital signatures based on public-key cryptography. With the advent of quantum computers, public-key cryptography becomes vulnerable since Shor's algorithm [1] was proven to solve the discrete logarithm problem that is the essence of public-key cryptography. Hence, scientists have already started to work on alternatives that could potentially provide strong security in the existence of quantum and traditional computers. Undoubtedly, there is not a universal quantum-proof solution that can deliver a unique solution to all the existing systems. Therefore, each of them requires separate investigation and design decisions based on the analysis of system limitations. DNS, an essential enabler of internet browsing, should also receive scrutiny. In this research, we investigate a possible application of quantum-resistant hash-based signatures, namely the Extended Merkle Signature Scheme in combination with Merkle Tree on the security of the DNS. Although hash-based signature schemes and Merkle Tree are known for quite a long, the signature size and signature verification time (for hash-based signatures) have been the major hindrance for their large-scale adoption. On the other hand, DNS messages have a limitation on the size, signing and verification speed. Unfragmented DNS message size is limited to the MTU (1500 bytes) if TCP connection is used; otherwise, UDP message size is limited to 512 bytes. Additionally, signing and resolution operations also need to be at least as fast as they are now. Therefore, this creates additional challenges to make DNS quantum-proof with little impact on its performance and signature size. Nevertheless, what would be the relative impact of using a quantum-proof approach on DNS signing, verification and message size? Recent studies mainly focus on the replacement of currently used PK algorithms in DNSSEC such as RSA with newly developed quantum-safe algorithms that rely on Lattice-based, Multivariate or Hash-based Cryptography [2]. On the contrary, we experiment with a combination of hash-based signature schemes (XMSS) and Merkle Tree authentication with DNS while incorporating its limitations and requirements. Merkle Tree is a simple binary tree of hash nodes that provides a quantum-resistant security mechanism. Hash-based signature schemes are based on the Merkle Tree, and even though both exist for a while, they possess certain disadvantages which prevent their mass usage. The main contributions of this research are to give a solid idea about the trade-offs in the proposed quantum-proof approach and analyse its impact on DNS performance.

1.1 Report organization

The remainder of this report is organized as follows. In Chapter 2, we give background information on DNS, DNSSEC and existing Hash-based Signature Schemes. Later, we introduce the approach and research questions as well as relevant literature for this thesis. Chapter 4 provides the design for the approach and answers the first research question. Chapter 5 describes methodology and prototype details. Results from prototype are followed by evaluation in Chapter 6. Finally, in Chapter 7, conclusions and recommendations are given.

Chapter 2

Background

2.1 DNS

Domain Name System (DNS) is an essential network protocol to access and retrieve online web resources. It provides a mapping of domain names such as *cloudflare.com* to an IP address so that the browsers can load the information provided by the endpoint located in that IP address. This mapping process, called DNS lookup, involves two types of globally distributed DNS servers: *recursive* and *authoritative* DNS servers. Recursive DNS servers, also referred to as DNS resolvers, are operated by the local Internet Service Provider (ISP) and responsible for recursively sending the DNS queries requested by the user (browser) to authoritative DNS servers. Initially, the recursive DNS server attempts to lookup for the domain name to IP address mapping in its cache of DNS records which is updated regularly once TTL values of DNS records expire. The existence of the cache memory mainly serves the purpose to reduce the network traffic by avoiding the whole recursive querying process for the Domain Names that have been recently accessed (non-expired TTL value). If the mapping does not exist in the cache, the recursive DNS server redirects the query to the Root Server. There are, in total, 13 globally distributed Root Server addresses, whereas actual instances of Root Servers are more than that thanks to the anycast addressing. At the time of writing, the Root Domain contains around 1500 Top Level Domains (TLDs) [3], which are classified as generic top-level domains (gTLDs) such as ".com", and country-code top-level domains (ccTLDs) such as ".nl". The Root Server responds with the address of the corresponding TLD DNS servers (".com" in the case of cloudflare.com) for further querying. Recursive DNS server sends the same query to the specified TLD server, and TLD server responds with the address of authoritative name server for cloudflare.com. Finally, querying the authoritative name server for ”cloudflare.com” will result in retrieving the A or AAAA records that contain the IPv4 or IPv6 address respectively of the server hosting ”cloudflare.com”. The user will retrieve the resource hosted by the provided endpoint. The summary of this process is depicted in Figure 2.1. Note that if the DNS resolver has already cached the corresponding DNS Resource Record (RR), the entire process is shorter as shown in Figure 2.2. After the expiration of the TTL value of RR, the extensive querying process has to be repeated. It ensures that any changes to the RRs are regularly communicated to the DNS resolvers.

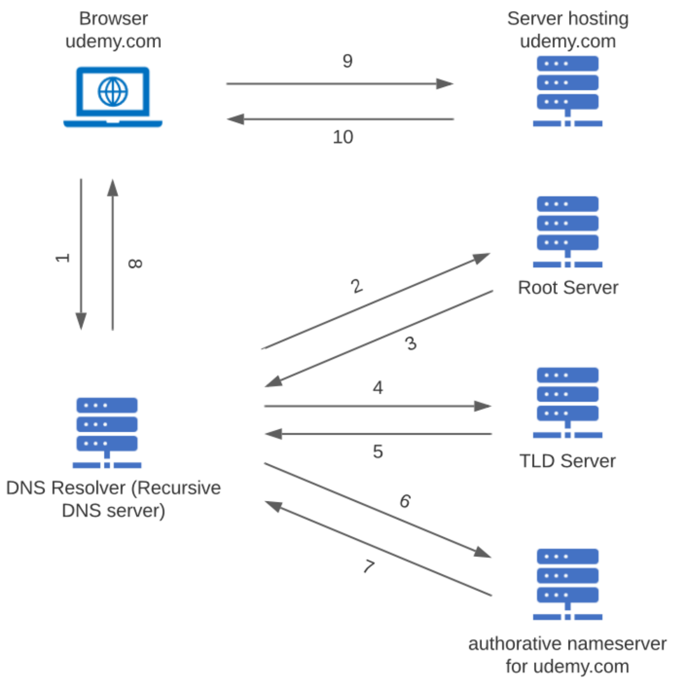


Figure 2.1: DNS complete lookup process

Challenges and Limitations. The initial design of DNS was not intended to satisfy major security and privacy concerns such as the integrity and authenticity of the communication between DNS resolvers and authoritative servers. Thus, a variety of DNS attacks have been launched against the resolvers as well as the name servers throughout the years. Cache Poisoning attack [4] is one type of attack that exploits the lack of authentication of the DNS records. As mentioned above, DNS Resolvers preserve the cache of DNS records for faster references. In a simple Cache Poisoning attack, an attacker tries to inject a malicious IP address into a name server cache. To generate the attack, the intruder needs to request a web resource that has not been cached by the resolver so that the lookup process takes place. Once the resolver starts the lookup process, it sends the query to the authoritative name servers after receiving a response from the Root Server. Meanwhile, the intruder tries to spoof the IP of authoritative name servers and floods the DNS resolver with responses that include malicious IP addresses. Resolvers accept the first response only if the below criteria match:

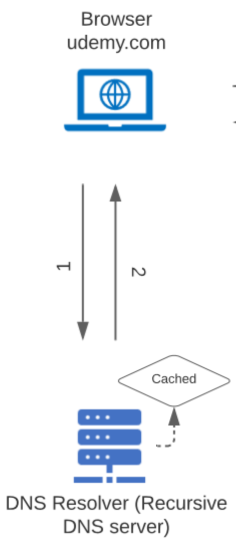


Figure 2.2: DNS lookup process for a cached record

- Response ID should be identical to query ID
- Response should arrive at the same UDP port as it was originated from

At the early development of DNS, these two criteria were poorly designed. The query ids were incremented by one; hence, it was sufficient for the intruder to observe traffic and try to send the response by increasing the ID. On the other hand, the UDP source port was also fixed as 53. These issues were attempted to be mitigated by randomizing the IDs and source port which resulted in a 2^{16} number of possibilities per each criterion. It should be noted that these mitigation techniques did not necessarily resolve the issues but made it slightly challenging and time-consuming. Moreover, classic DNS packets are restricted to 512 bytes. With extension mechanisms for DNS (EDNS(0)) [5], the accepted DNS packet size can be up to 64kB. On the other hand, the Maximum Transmission Unit (MTU) size is 1500 Bytes for IPv4 packages. Typical DNS responses are generally short hence unfragmented. If EDNS(0) is supported, attackers can often cause fragmentation by registering a maliciously crafted subdomain [6] to make the packet size larger than MTU. Attacks that exploit the limitations of DNS urged the community to find a new way of securing DNS – DNSSEC (DNS Security Extensions) which is discussed in the next section.

2.2 DNSSEC

DNSSEC is designed to meet the authentication and integrity requirements of DNS by signing record sets using public-key cryptography. A cryptographical signature is added to each DNS response and transmitted to the resolvers.

PKI Signing and Verification. Authoritative name servers handle the signing process with the help of zone operators. Signature generation involves a few sets of records which are explained below in detail.
RRset – Resource Record Set
Securing each DNS Zone starts with grouping identical record types (e.g. all AAAA records) for the same domain name into RRsets. In fact, it is the RRsets that is signed instead of the single record. Hence, there could be multiple RRsets in a zone such as one for AAAA records and one for MX records.

RRSIG – Resource Record Signature
Each RRset is signed using a private portion of the zone signing key (ZSK) maintained in each zone. Signed RRsets stored in RRSIG records in the name servers controlled by zone operators. The public portion of ZSK is used during the verification process and needs to be accessible by the resolver. Therefore, along with the RRset and RRSIG records, the zone operator also provides a public portion of ZSK in the DNSKEY record. These three sets of information are essential but insufficient to trust the DNS records.

KSK – Key-Signing Key
To trust all the records in the zone, it is crucial to verify public ZSK in the DNSKEY record [7]. To do that, another pair of keys – KSK is generated, and a public portion of it is stored in another DNSKEY record. A private portion of the KSK is used to sign public ZSK as well as its public KSK. The signed RRset of DNSKEY records is stored in a new RRSIG record. Thus, RRset of DNS records and corresponding RRSIG record; RRset of DNSKEY records (Public ZSK and KSK) and corresponding RRSIG record will be sent by the name server to resolvers. Now, the public KSK of the zone should also be verified since signing public KSK with its own private KSK does not provide any additional security (man in the middle can replace the signed DNSKEY RRset record using a different KSK pair). Because zones are interconnected and usually operate dependently as shown in figure 2, DNSSEC introduces a new record called Delegation Signer (DS) which is discussed below.

DS – Delegation Signer
The purpose of the Delegation Signer is to establish the chain of trust among parent and child zones and provide a means for verification of public KSKs for resolvers. The child zone operator provides the parent zone with its public KSK by hashing the DNSKEY record, which contains public KSK, and the parent zone publishes it as a DS record. Whenever a resolver receives a reference from a parent zone to a child zone, the DS record of the child zone is also provided. This way, the resolver has a

means of verifying the public KSK by hashing the DNSKEY record sent by the child zone and comparing it to the DS record retrieved by the parent zone. It could be argued that the way public KSK is used to build up trust, it could have been done the same way with public ZSK without introducing a new KSK pair. The underlying reason for this design is due to the cryptographical strength and the limitation of DNS message sizes. It is desired to have a smaller ZSK key and signature with regular key changes to minimize the exchanged data size. Regular key renewals ensure that compromised ZSK keys (particularly when the key size is small) are not preserved for a long period. Swapping out DS records is expensive since it requires delegating the key to the parent. Thus, if ZSK was used in the DS record, regular key rollovers should have been performed which is not desirable due to the complex multi-step process performed by the parent and child zone. Therefore, to decrease the frequency of key rollovers without compromising the security, longer KSK is used. Hence, ZSK is smaller than KSK and easier to perform key rollovers as it only involves a particular zone. On the other hand, because KSK is relatively larger, the key rollover does not have to be performed regularly. Ultimately, the relation between small key size – frequent key rollovers and large key size – rare key rollovers are achieved with ZSK and KSK design. In the above design, the DS record also needs to be trusted. Therefore, like other records, the DS record is signed by the private ZSK of the parent. The whole verification process is continued until the root DNS server. Root DNS server signs the DS record of the child but there is no means to verify the public KSK of the root server as there is no parent DS record for the root server. Nevertheless, DNS resolvers are pre-configured with the public KSK of the root server and trusted by default. The summary of the Signing and Verification Process for www.cloudflare.com is as follows:

1. Root DNS Server (signing)

   - signs the DS record for ".com" TLD zone with root zone's PrvtZSK – RRSIG1

   - signs RRset of DNSKEY records (containing root zone's PubKSK and PubZSK) with its PrvtKSK – RRSIG2

   - sends RRSIG1, corresponding unsigned DS record along with the referral for the authoritative name servers for the ".com" zone. Only after the resolver's separate query request for root DNSKEY, the Root Server will send the RRSIG2 and corresponding RRset of the root zone's DNSKEY records (DNSKEY PubKSK and DNSKEY PubZSK).

2. Recursive DNS server (verification)

   Zone verification:

- verifies root zone's DNSKEY PubKSK by comparing it to already obtained PubKSK through pre-configuration done by e.g. server's operating system's vendor.

Record verification:

- verifies RRset of root zone's DNSKEY records (PubKSK and PubZSK) by decrypting the RRSIG2 using preconfigured PubKSK (trusted).
- verifies DS record for ".com" zone by decrypting RRSIG1 using PubZSK which was verified above.
- sends the same query to a ".com" zone for www.cloudflare.com. ".com" TLD server (signing)

3. ".com" TLD server (signing)
   - signs the DS record for the "cloudflare" zone with its PrvtZSK – RRSIG1
   - signs RRset of DNSKEY records (containing ".com" zone's PubKSK and PubZSK) with its PrvtKSK – RRSIG2
   - sends RRSIG1, corresponding unsigned DS record for "cloudflare" zone along with the referral for the authoritative name servers for the "cloudflare" zone. Only after resolver's separate query request for DNSKEY, ".com" TLD server will send the RRSIG2 and corresponding RRset of ".com" zone's DNSKEY records (DNSKEY PubKSK and DNSKEY PubZSK).
4. Recursive DNS server (verification)
   ".com" Zone verification:
   - verifies the zone by comparing the hashed value of the DNSKEY PubKSK record to the previously obtained DS record for the "com" zone from the Root Zone.

   Record verification:

   - verifies RRset of "com" zone's DNSKEY records (DNSKEY PubKSK and DNSKEY PubZSK) by decrypting the RRSIG2 using "com" zone's PubKSK (which was trusted as a result of Zone verification process).
   - verifies DS record for "cloudflare" zone by decrypting RRSIG1 using PubZSK which was verified above.
   - sends the same query to a "cloudflare" zone for www.cloudflare.com

5. “cloudflare” authoritative server (signing)
    - signs RRset of A or AAAA records for “cloudflare” zone with its PrvtZSK – RRSIG1
    - signs RRset of DNSKEY records (containing “cloudflare” zone’s PubKSK and PubZSK) with its PrvtKSK – RRSIG2
    - sends above-signed records (RRSIG1 and RRSIG2), corresponding RRset of A/AAAA records for “cloudflare” zone and RRset of zone’s DNSKEY records (DNSKEY PubKSK and DNSKEY PubZSK).
6. Recursive DNS server (verification)
   “cloudflare” Zone verification:
    - verifies the “cloudflare” zone by hashing the DNSKEY PubKSK record and comparing the hashed value to the previously obtained DS record for the “cloudflare” zone from the “com” zone.

   Record verification:
    - verifies RRset of “cloudflare” zone’s DNSKEY records (DNSKEY PubKSK and DNSKEY PubZSK) by decrypting the RRSIG2 using “com” zone’s PubKSK (which was trusted as a result of Zone verification process).
    - verifies RRset of A or AAAA records by decrypting RRSIG1 using PubZSK which was verified above.

Challenges and Limitations. While DNSSEC addresses the limitations of DNS by providing authenticity and integrity of the DNS records via digital signatures, this additional security comes with a price of an increase in DNS responses size. Attackers have recently leveraged this increased DNS response size for DNS Amplification attacks [8]. According to the findings by Nexusguard Research [9], a big portion of the domains that were abused in DNS Amplifications had indeed deployed DNSSEC. Various research has been conducted to tackle the size limitation of DNSSEC by replacing the underlying digital signature algorithms [8], [10], [11]. Root zone KSK rollover is another challenge in DNSSEC. Since the resolvers are manually configured with the Root zone’s KSK, in case of the disruption that might require the renewal of the root KSK, it will cause manual intervention that could be very prone to errors.
DNSSEC relies on digital signatures and the power of signatures on their own is limited to the security of the underlying cryptographic algorithms. That is, any newly introduced weakness to the digital signature algorithms (particularly PKI) will potentially threaten DNSSSEC infrastructure and require undesirable drastic changes.

The strength of PKI, on the other hand, lies in the “unsolvable” discrete logarithm problem by classical cryptography and computers. Quantum algorithms such as Shor’s algorithm [1] are highly specialized to solve this “unsolvable” problem. With the construction of large-scale quantum computers on which quantum algorithms could be eventually run, currently deployed public-key algorithms including RSA, ECDSA and EdDSA will be breakable [12]. Thus, new security mechanisms that will be resistant to quantum computers are being researched intensively. In particular, constraints such as DNSSEC response size, signing and verification time need to be considered while developing quantum-proof security mechanisms to make it effective for DNSSEC.

2.3 Hash-based Signature Schemes

Public-key cryptography relies on the number-theoretic problems such as Pell’s equation, factoring and discrete log. In fact, quantum computers are shown to solve the factoring and discrete log problem in polynomial time [1], and Pell’s equation in exponential time [13]. As the traditional digital signatures are based on public-key cryptography, it is critical to come up with new post-quantum signature mechanisms that promise strong security in the existence of quantum as well as traditional computers. One of the potential post-quantum signature schemes is Hash-based signatures. Unlike conventional digital signature algorithms (RSA, DSA, ECDSA), the security of hash-based signatures does not rely on the difficulty of factoring large composite integers and computing discrete logarithms [13]. It is one of the reasons that lead cryptographers to believe in the quantum immunity of hash-based signatures. Hash-based signatures use cryptographic hash functions which take input strings of arbitrary length and outputs strings of short, fixed-length called message digest [14]. Those hash functions must satisfy three criteria [15] to be considered secure:

- Preimage resistance – given a hash function h and hash value v: v = h(m) where m is the unknown message, it ought to be hard to find the message m that yields the hash value v.

- Second preimage resistance – given a message m and hash function h, it ought to be hard to find another message m’ such that h(m) = h(m’) where m $\neq$ m’.

- Strong collision resistance – given only a hash function h, it ought to be hard to find two messages m and m’ such that that h(m) = h(m’) where m $\neq$ m’

The difference between collision resistance and the second preimage is whether the message is known or arbitrary. Collision resistance itself implies the second preimage resistance: an attacker can pick any message m and try to compute the second preimage message m' which should yield the same hash value as the message m. Hash-based digital signatures rely on the collision resistance or second preimage resistance of the hash functions. This is considered as a minimum requirement for the existence of the digital signature scheme [10]. Hash-based signature schemes are potentially good candidates for the quantum era, and each existing scheme has already been undergone some improvements and changes. The hash-based signature schemes explained below make use of hash function g denoted as

$$g : \{0,1\}^* \rightarrow \{0,1\}^*$$

and one-way function

$$f : \{0,1\}^n \rightarrow \{0,1\}^n$$

Additionally, each scheme has different security parameters which are discussed in a more detailed manner.

2.3.1 Lamport One-time Signature Scheme (L-OTS)

One of the early proposals for hash-based signature schemes is L-OTS [16]. The security parameter of L-OTS is a positive integer n. The size of the keys, as well as the signature, is expressed in terms of this security parameter. The essential idea behind the scheme is that the private and public key must only be used once to sign a message.

Key pair generation. Let's denote the private key as PrvtK. It is composed of 2n strings or n pairs; each consists of n bits uniformly picked at random. The first element of *i*th pair is denoted as x_i[0] and the second element of *i*th pair is denoted as x_i[1]:

$$PrvtK = ((x_{n-1}[0], x_{n-1}[1]), ..., (x_1[0], x_1[1]), (x_0[0], x_0[1])) \in R(0,1)^{(n,2n)}$$

The signer needs to store all the pair elements to compute PrvtK. Since each element is n bit string, this makes size of PrvtK equal to $2n * n = 2n^2$. Let's denote public key as PubK. Similarly, PubK is composed of 2n bit strings or n pairs; each consists of n bits but instead of picked at random, it is the one-way function of the PrvtK pairs with a length n.

$$PubK = ((y_{n-1}[0], y_{n-1}[1]), ..., (y_1[0], y_1[1]), (y_0[0], y_0[1])) \in R(0,1)^{(n,2n)}$$

where

$$y_i[j] = f(x_i[j]), \quad 0 \leq i \leq n-1, j = 0, 1.$$

Thus, both PrvtK and PubK has a size $2n^2$. Moreover, PubK also requires 2n evaluation of f one-way function.

Signature generation. The first step towards signing a message M is computing the message digest d using the hash function g which was defined previously:

$$d = g(M)$$

A signature will be computed on the digest d as follows: if the *i*th bit in the digest is 0 then the signature will be the first element of the *i*th pair in PrvtK or if the *i*th bit in d is 1 then the signature will be the second element of the *i*th pair in PrvtK. In another word, *i*th bit in the digest determines the element of the *i*th pair to be used for the signature.

$$Sig = (x_{n-1}[d_{n-1}], ..., x_1[d_1], x_0[d_0]) \ \in R(0,1)^{(n,n)}$$

Signature Sig has the length of n^2, and no evaluations of function f are required. The sender provides signature Sig, $PubK$ and plain text message M for verification which is explained below.

Signature verification. First step of verification is to compute the message digest using the same hash function as above. Then each string in signature denoted with small sig will be fed to the one-way function f and following will be checked:

$$f(sig_{n-1}), \dots, f(sig_0) = (y_{n-1}[d_{n-1}], \dots, y_1[d_1], y_0[d_0]) \ \in R(0,1)^{(n,n)}$$

If we substitute sig_i with above signature generation formula and public key with its generation formula, we get identical formulas:

$$(f(x_{n-1}[d_{n-1}]), \dots, f(x_0[d_0])) = (y_{n-1}[d_{n-1}], \dots, y_0[d_0]) = (f(x_{n-1}[d_{n-1}]), \dots, f(x_0[d_0]))$$

Unlike signature generation, signature verification requires n evaluations of function f.

Challenges and Conclusion. Although signature and key generation in the L-OTS scheme can be considered efficient, the main disadvantages of L-OTS are the large key and signature sizes [17], and impractical one-time key usage. The signer has to use the PrvtK and PubK only once so after each signature, the signature generation process needs to be repeated. To tackle the large signature size issue, Winternitz offered his modified one-time signature scheme which is explained in the next subsection.

2.3.2 Winternitz One-time Signature Scheme (W-OTS)

The main goal of W-OTS is to achieve a smaller signature and key size. Unlike L-OTS where each bit in the message digest is signed using the corresponding private key element, W-OTS uses a compact approach where several bits in the message

digest will be signed using one private key element. To accomplish this, the scheme introduces a new security parameter w to denote the number of bits to be signed at once and following changes to key pair generation and signing.
Key pair generation. In the W-OTS scheme, w is the security parameter that shows the number of bits in the message digest d to be signed simultaneously. The number of private key elements that form the PrvtK is reduced by a few factors depending on the choice of security parameter w and determined as such:

$$l = l_1 + l_2$$

where

$$l_1 = \lceil n/w \rceil, \quad l_2 = \lceil (\lceil log_2 l_1 \rceil + 1 + w)/w \rceil$$

l_2 here shows the number of private key elements to sign the checksum of the message digest. Thus, PrvtK is composed of l key elements, each uniformly distributed at random with length n.

$$PrvtK = (x_{l-1}, \ldots, x_1, x_0) \; \in R(0,1)^{(n,l)}$$

PubK is computed by applying the one-way function f to each PrvtK element $2^w - 1$ times.

$$PubK = (y_{l-1}, \ldots, y_1, y_0) \; \in R(0,1)^{(n,l)}$$

where

$$y_i = f^{(2^w-1)}(x_i), \quad 0 \leq i \leq l-1.$$

Both PrvtK and PubK have the size $n * l$. PubK generation requires $l * (2^w - 1)$ computation of function f.
Signature generation. Signature generation starts with hashing the message to produce the message digest d. As mentioned above, in this signature scheme, w bits are signed simultaneously. Therefore, bits in message digest d are grouped into w bits which results in l_1 blocks assuming the length n of the message digest is divisible by w; otherwise, extra zero bits are appended to the digest.

$$d = (d_{l_1-1}, \ldots, d_1, d_0)$$

Additionally, checksum of the message digest is also calculated since actual message with the checksum is signed, not the single message:

$$C = \Sigma_{i=0}^{l_1-1}(2^w - d_i)$$

where d_i is binary to decimal conversion of each w bits group in message digest. Therefore, the maximum value that d_i could get is $2^w - 1$. The checksum C will be converted back to binary representation and similar to d, divided into blocks of w bits which are appended to the message digest d:

$$M = d_{l_1-1}||\ldots||d_1||d_0||c_{l_2-1}||\ldots||c_1||c_0 = (m_{l-1},\ldots,m_1,m_0)\ \in R(0,1)^{(w,l)}$$

Finally, signature is generated by applying m_i times one-way function f on PrvtK element x_i.

$$Sig = (sig_{l-1},\ldots,sig_1,sig_0)$$

where

$$sig_i = f^{m_i}(x_i), \quad 0 \le i \le l-1,\ 0 \le m_i \le 2^w - 1$$

Since each x_i consists of n bits, this makes signature size $l * n$ long. Additionally, it requires at most $l * (2^w - 1)$ computations of f function since maximum value of m_i could be $2^w - 1$.

Signature Verification. Signature $Sig = (sig_{l-1},\ldots,sig_1,sig_0)$ verification involves calculation of message M as above and comparison of public key as following:

$$y_i = f^{2^w-1}(x_i) = f^{2^w-1-m_i}(s_i), \quad 0 \le i \le l-1,\ 0 \le (m_i) \le 2^w - 1$$

Thus, applying the f function on blocks of signature as above would yield PubK if the signature is valid. Signature verification, similar to generation, requires at most $l * (2^w - 1)$ computations of f. An improved version of W-OTS – W-OTS+ reduces the security requirement of the underlying hash function to a second pre-image resistance [18]. Instead of simple one-way f function to be iterated, function T is defined as such:

$$T^i(x,\mathbf{r}) = f(T^{i-1}(x,\mathbf{r}) \oplus r_i)$$

Each iteration is the one-way function of the output from the previous iteration XOR-ed with a random string r_i in $r = (r_1,...,r_j)\ \in (0,1)^{nj}$.

Challenges and Conclusion. Figure 2.5 summarizes the key and signature size, generation and verification time and complexity. It shows that the key as well as the signature size of W-OTS is smaller than L-OTS. However, this comes at the price of an increase in the computational cost of key and signature generation and verification. Table 1 shows the relation between time and Winternitz parameter w. A higher value of w would allow signing more bits simultaneously and result in a shorter signature and key size (smaller l decreases the size linearly). Nevertheless, this results in slow key and signature generation and signature verification (exponential increase). Therefore, the choice of w should be done carefully considering the trade-off between key/signature size, signature generation as well as verification time. Moreover, W-OTS on its own is still inefficient as PrvtK must only be used only once, and new random key elements have to be generated each time after signing.

2.3.3 Merkle Signature Scheme (MSS)

Even though W-OTS improves key and signature size problems, key management remained unsolved. In the above schemes, the new signature generation requires the new key generation and exchanging long public keys, which are complex and costly processes [17]. Therefore, Merkle introduced a new way of key management in which keys and their sizes are reduced [17], [19]. With MSS, instead of the long PrvtK, a key pair is used to generate a new signature, using any of the one-time signature generation schemes. It means a predefined number of messages can be signed with the same PubK, thus, reducing the frequency of public key generation. Public key elements of a key-pair are arranged in a binary tree to form a PubK of which verification is required along with signature verification. The following explains each process in detail.

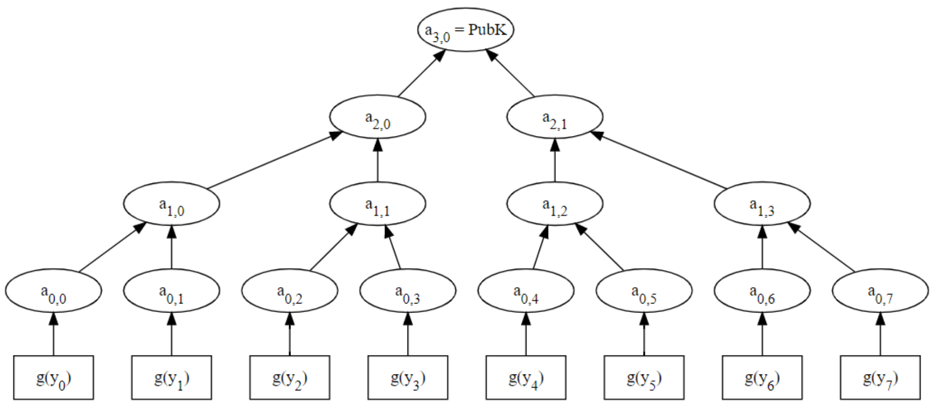


Figure 2.3: Merkle's Hash Tree

Key Pair generation. Similar to key pair generation in L-OTS and W-OTS, 2^H number of one-time private and public key pairs (x_i, y_i), $0 \leq i < 2^H$, $H \geq 2$ are generated. The number H defines the height of the Merkle's binary hash tree. Figure 2.3 demonstrates the tree structure with height $H = 3$. The leaf nodes of the tree are hash values of public key elements from the generated pairs: $g(y_i)$. The inner nodes are hash of the concatenation of left and right children:

$$a_{j,h} = g(a_{h-1,2j} \mid\mid a_{h-1,2j+1}) \quad 0 \leq h \leq H, \; 0 \leq j < 2^{H-h}$$

h shows the level of node a, counted from leaf (level 0) node to the node itself and j shows the node number counted from left to right in the level h. PrvtK is the concatenation of all key pairs (x_i, y_i) and PubK is the root of the tree. Thus, PubK generation requires $2^{(n+1)} - 1$ calculation of hash function g.
Signature generation. Instead of using the PrvtK to sign the message, only one of the one-time private signature key pairs is used per message. Therefore, with MSS,

at most 2^H messages can be signed. Assume the *k*th key x_k is picked for the one-time signature generation that uses any signature scheme mentioned previously. MSS signature will be composed of the one-time key index k, corresponding public key element y_k, generated one-time signature sig_{ots} and the authentication path A_k from public key element y_k to the root of the tree PubK:

$$Sig = (k, y_k, sig_{ots}, A_k)$$

where

$$A_k = a_0, \ldots, a_{n-1}$$

Authentication path allows efficient reconstruction of Merkel hash tree to validate the root of the tree – PubK without revealing all the key pairs and exchanging the PubK. To reconstruct the tree, only n nodes need to be known. An example authentication path for $k = 2$ is shown in Figure 2.4. The one-time private key element to be used for signing is x_2 and public key element to be shared is y_2 . To calculate and verify the root of the tree from y_2, it is sufficient for receiver to know the nodes $a_{0,3}, a_{1,0}, a_{2,1}$. Thus, the authentication path is $A_3 = (a_{0,3}, a_{1,0}, a_{2,1})$. Signature

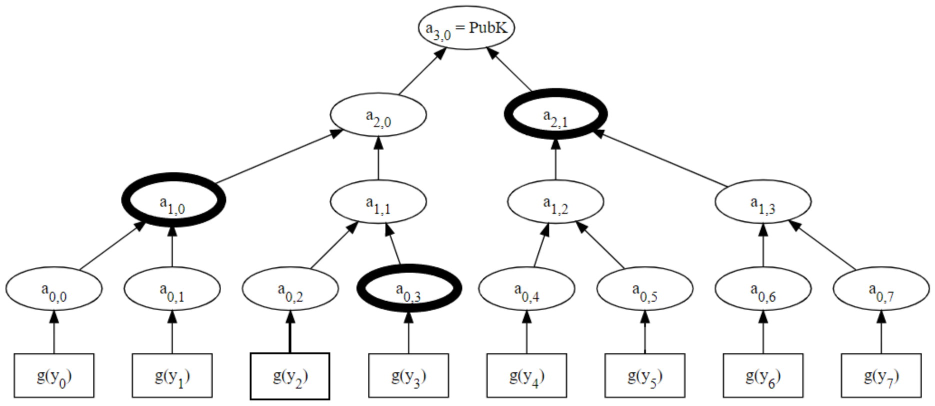
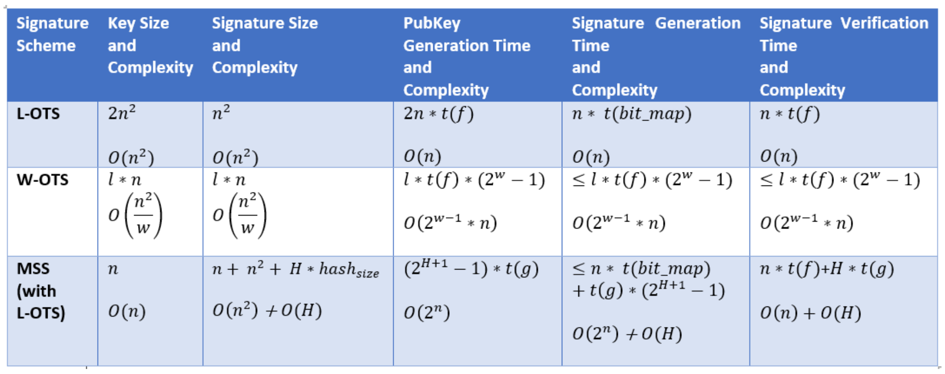


Figure 2.4: Authentication path for k=2

generation requires one-time signature generation sig_{ots} and computation of an authentication path, which might require entire tree construction - $2^{H+1} - 1$ evaluations of hash functions at worst if no node is pre-stored. Note that algorithm choice for authentication path mainly concerns the trade-off between time and storage capacity [20], [21], [22].

Signature verification. Verification has two parts. The first part is verifying sig_{ots} using a verification algorithm of the corresponding OTS and y_k. Once the signature has been verified, using auth path A as well as y_k, the root of the Merkel tree will be computed. Note that the receiver knows the PubK and will compare the output from the calculation of the root of the tree to the PubK. PubK Verification requires only H hash operations and sig_{ots} depends on the chosen OTS.

Challenges and Conclusion. Merkle OTS has a clear advantage of signing many signatures without generating a new PubK every time. However, this leads to a trade-off between computation time and storage. Computation time to build the Merkle's Hash Tree includes time to generate 2^H one-time private (signature) keys and $2^{H+1} - 1$ evaluations of hash functions, one evaluation per node. Construction of the entire tree is a costly operation and needs to be done irregularly. To generate the signature, an authentication path consisting of H nodes need to be sent. In case the nodes are not pre-stored, the node computations must be done again for each signature. On the other hand, storing all pre-computed nodes might require a large storage capacity. Therefore, there is a need for an efficient technique to compute the authentication path on demand (either due to an update in one-time key pairs or new signature generation) without storing too many nodes. In literature, this problem is called the Merkle Tree Traversal problem, and several algorithms [20], [21], [22] have been proposed to optimise the time complexity of authentication path calculation using less memory capacity. Moreover, MSS's signature size is also a bottleneck and has been the major downside for the common use and adaption.



Signature Scheme	Key Size and Complexity	Signature Size and Complexity	PubKey Generation Time and Complexity	Signature Generation Time and Complexity	Signature Verification Time and Complexity
L-OTS	$2n^2$ $O(n^2)$	n^2 $O(n^2)$	$2n * t(f)$ $O(n)$	$n * t(bit\_map)$ $O(n)$	$n * t(f)$ $O(n)$
W-OTS	$l * n$ $O\left(\frac{n^2}{w}\right)$	$l * n$ $O\left(\frac{n^2}{w}\right)$	$l * t(f) * (2^w - 1)$ $O(2^{w-1} * n)$	$\leq l * t(f) * (2^w - 1)$ $O(2^{w-1} * n)$	$\leq l * t(f) * (2^w - 1)$ $O(2^{w-1} * n)$
MSS (with L-OTS)	n $O(n)$	$n + n^2 + H * hash_{size}$ $O(n^2) + O(H)$	$(2^{H+1} - 1) * t(g)$ $O(2^n)$	$\leq n * t(bit\_map) + t(g) * (2^{H+1} - 1)$ $O(2^n) + O(H)$	$n * t(f) + H * t(g)$ $O(n) + O(H)$

Figure 2.5: Space and Time comparison of one-time signatures

2.3.4 Extended Merkle Signature Scheme (XMSS)

XMSS is standardized, the most efficient existing hash-based signature scheme [23]. It introduces several improvements on MSS such as *1. reduced signature size*, *2. reduced security requirement – from collision resistant to the second pre-image resistant hash-function*, *3. forward security property* and *4. unforgeability under Chosen Message Attacks*. XMSS achieves these advantages with three major design choices made on top of the MSS. First, it uses W-OTS or W-OTS+ (preferably) as a one-time signature scheme. As discussed above, W-OTS already outputs a smaller signature. Additionally, XMSS signature (or MSS if used with W-OTS) does not need to include the one-time public key since it can be calculated from the W-OTS signature and if the root of the tree is verified successfully by using that one-time public key, it indicates the authenticity of the signature and one-time public key. This reduces the signature size from $\approx 2n^2 + nlog_2N$ to $\approx n^2 + nlog_2N$, roughly a factor of 2 [23]. Second design choice is the usage of bitmasks in tree node calculation. The hash function inputs the bitmasks along with the child nodes to calculate the parent node in Merkle's binary hash tree:

$$a_{h,j} = g((a_{h-1,2j} \oplus bitmask_{left,h}) || (a_{h-1,2j+1} \oplus bitmask_{right,h})),\ 0 \le h \le H,\ 0 \le j \le 2^{H-h}$$

This provides the second improvement mentioned above which allows the replace-

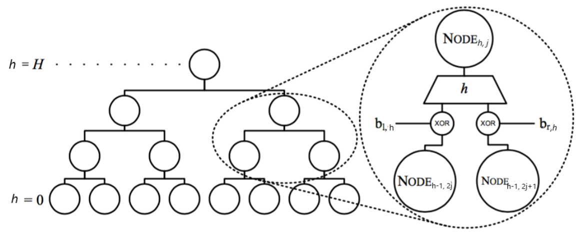


Figure 2.6: XMSS tree [24]

ment of collision-resistant hash functions [18], [25]. Third design choice introduced in XMSS that is different from MSS is the leaf node construction. The leaf nodes in XMSS are computed by constructing another XMSS tree called L-tree in contrast to the MSS where it is simply the hash of the one-time public key. The leaves of the L-tree are the bit strings of the corresponding one-time public key. In case is not the power of two, the node that does not have the right sibling will be levelled up until another node needs the right sibling. The nodes are calculated the same as in XMSS. Each L-tree will have the same bitmasks but different from the one used in the XMSS tree. That means additional bitmasks are required per tree level. The root of the L-trees will be the leaf nodes of the XMSS tree. This also adds up to the improvement towards the reducing the security requirement to second pre-image resistant hash functions. Forward security property states that even a one-time private key is compromised, all the previously generated signatures are valid. This statement holds for the key evolving schemes where each one-time key pair has a lifetime period T that will be expired after a signature is generated using that key pair. XMSS is forward secure [18] with a lifetime period T showing the maximum number of signature generations. Signature generation invokes key update functions and the index keeps track of generated signatures. Since XMSS should keep track of generated signature and key pair, it is considered a stateful signature scheme. Any machine that is running the algorithm needs to preserve the state of the algorithm in a reboot.

Chapter 3

Problem Statement

3.1 Research Questions

Hash-based signatures are considered one of the quantum robust methods along with lattice-based as well as code-based cryptography. The focus of the research is, however, solely on hash-based signatures schemes and specifically their application on DNSSEC. While examining quantum robust signature options as potential future alternatives, the challenges of DNSSEC and its signature requirements must be taken into account. As mentioned earlier, DNSSEC introduces an overhead of an increase in response size due to the attached signatures, and this potentially can lead to fragmentation issues and an increase in amplification attacks. Thus, while considering the acquisition of the quantum-safe digital signature algorithms, one must consider the signature size. Moreover, in [2], desirable values for DNSSEC signature and key size, as well as signature verification and generation speed are approximated as part of requirements for quantum-safe algorithms. To meet the requirements, Westerbaan, in his private research statement (can be provided upon request), suggested to form Merkle trees out of DNS records, and instead of signing each record, only the root of the tree will be signed using hash-based signature algorithms such as XMSS. Similarly, [26] suggested the same approach except labelling the root of the Merkle tree as-synthesized ZSK. In this research, we want to explore the impact and effectiveness of this approach concerning DNSSEC requirements. We aim to answer the following questions:

- What are the important variables to be considered if we use Merkle tree Authentication and sign the root of the tree with XMSS in DNSSEC?
- What are the impacts of those variables on signature size, signature generation and verification speed in DNSSEC?
- How will grouping domains in different Merkle trees depending on the update frequencies impact the verification speed and frequency of tree updates?

3.2 Approach

We propose a two-level approach similar to the current design choice of DNSSEC with differences stated below.

Record authentication. The first level ensures the Record authentication (ZSK level). The main difference between the current and proposed approaches is that in the current design, RRsets are signed using public-key cryptography resulting in RRSIG records. In contrast, simple Merkle trees will be used where the DNS records are the leaf nodes, and the root node is the one that will be signed using a hash-based signature which is discussed below. Thus, neither private ZSK nor signature algorithms other than hash functions are used at this level. Leaf nodes in a tree might be grouped based on the domain popularity (if A or AAAA records) and record types (all NSEC records). The authentication path to the root will be the only information provided as a signature. Note that the approach of a grouping of records based on popularity (update frequencies) and record types will further be explored during the research.

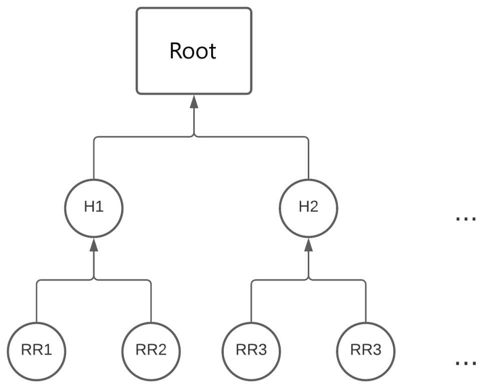


Figure 3.1: Merkle tree used for Record authentication.

Zone authentication. The second level entails Zone authentication (KSK level) where public ZSKs formed from the first layer (roots of Merkel trees) will be signed using hash-based signature algorithm XMSS. This corresponds to the generation of RRSIG record of public ZSK in current design of DNSSEC. The root of XMSS tree can be named as Public KSK and verified along with verification of ZSK.

We propose the implementation in three main steps. In step one, we will analyse the records in the ".nl" TLD zone provided by *SIDN Labs* to identify the popular domains and update frequencies for those domains. This will help us determine the important variables such as the amount of Merkle trees and their heights (depth). Secondly, we will model the signer and resolver accordingly. In the final step, we will do the benchmark test to check the impact of the variables on verification, signing speed

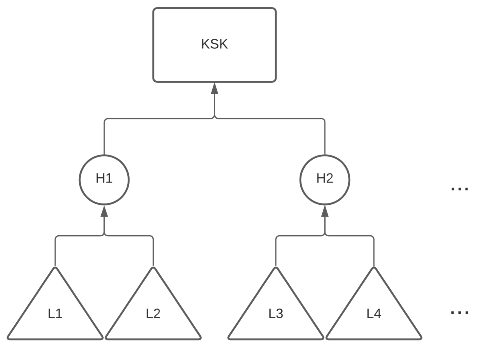


Figure 3.2: Merkle tree for Zone Authentication: XMSS trees as leaf nodes.

and signature size. We will also modify the records to see how updates impact the verification speed overall.

3.3 Literature Review

Hash-based signatures can be potentially used in DNSSEC to provide quantum-resistant security. Even though the idea of hash-based signatures dates back to the 20th century, it is not one of the commonly used signature schemes due to the relatively larger signature size. Applications of Merkle's tree are also limited to certification refreshal [27], broadcast authentication protocols [28], third-party data publishing [29], zero-knowledge sets [30], micropayments [31] and certificate transparency logs [32]. With the rising demand in Post-Quantum Cryptography, hash-based signatures are being revisited as a quantum-proof alternative to traditional signatures. Therefore, there is a lack of studies in securing protocols, particularly DNS with hash-based signatures. In general, we can divide relevant literature studies into three categories: The first category has investigated the *applications of hash-based signatures* on various areas other than DNSSEC. The second category studied *signature and performance improvements* on DNSSEC in a non-quantum era, and the third group studies the intersection of the first and second categories - *hash-based signatures on DNSSEC*.

Application Domains of Hash-based signatures. There have been several attempts to apply hash-based signatures in practice. Butin et al. [33] tried to integrate XMSS into the OpenSSL library to support XMSS based authentication in TLS and S/MIME protocols. They provided a common integration guideline for OpenSSL independent of the use cases as well as analysis and evaluations of XMSS in TLS and S/MIME. Analysis shows that the signature size is larger than RSA or DSA signature sizes as expected. Depending on the applications of TLS, a small signature size may or may not be the core requirement. For HTTPS, fast signing speed is more desirable than the small signature size [33] as the average size of a web page is already much higher, and technically, there is no limiting factor. This makes it suitable for XMSS^{MT} with more layers since more layers would speed up the signing process but increase the signature size. Additionally, more layers also make the state management more complicated [33]. In contrast, small signature size and fast signature verification are critical requirements for DNSSEC as mentioned above. Requirements and architectural decisions for S/MIME, however, differs largely from HTTPS and align with DNSSEC. Another study [34] measured the performance of post-quantum algorithms including SPHINCS^{+} from hash-based signature family on authentication in TLS 1.3. It was found out that SPHINCS^{+} performs particularly lower in the signing process. Kampanakis et al. [35] constructed similar research to check the viability of post-quantum signatures on X.509 certificates and protocols that use X.509 for authentication, namely TLS and IKEv2. A hybrid approach that utilizes traditional(RSA) and post-quantum signatures (Merkle tree with W-OTS) are used to construct the certificates: Post-quantum public key and the one-time signature algorithm are added as two extensions to the base certificate. The certificate is then encoded and signed using a one-time private key to produce the signature. This hash-based signature is added as a third extension to the base certificate and then signed using RSA or ECDSA private key. The research does not study the signing and verification speed of hash-based signatures but mainly the performance overhead of a protocol transmission caused by the certificate size which is shown to be less critical since TLS and IKEv2 use fragmentation of lengthy TLS records. [35]. A very different survey [36] was conducted on the analysis of hash-based signatures for IoT. IoT devices are performance and resource-constrained. Therefore, post-quantum security choices have to be made carefully considering the requirements. These are similar requirements to DNSSEC where performance constraints in IoT can map to a fast signature verification requirement in DNSSEC and resource constraints to limited response size (thus short signature size). The authors discuss six factors of hash-based signatures that are essential in decision making for IoT devices. Some of them could be potentially considered for DNSSEC.

Improvements on DNSSEC. Limitations of DNSSEC mentioned on page 5 led researchers to look for alternatives for the underlying architecture in DNSSEC. Van Rijswijk-Deij et al. [37], [10], [8] investigated the use of Elliptic Curve Cryptography(ECC) and Elliptic Curve Digital Signature Algorithm(ECDSA) in DNSSEC since the problems associated with DNSSEC appeared to stem from the default choice of RSA. The use of ECC has shown to have the disadvantage of increasing verification time on the resolvers side. [38] proposed to change the DNSSEC architecture completely by using Blockchain: eliminating chain of trust to yield easier key management and reducing the signature size. Nevertheless, none of these studies takes

the quantum challenges of DNSSEC into account.
DNSSEC with Hash-based signatures. Only two known pieces of literature exist on Hash-based signing in DNSSEC. Müller et al. [2] measured the performance of signature generation and verification using quantum-safe algorithms that are under standardization process by NIST [39]. First, they defined DNSSEC requirements for quantum-proof algorithms, namely signature size, optimal verification and signing speed. Only those algorithms that provide a smaller signature size than required were evaluated further for signing and verification speeds. From Hash-based signature schemes, three candidate algorithms (Sphincs$^+$-Haraka-128s, Picnic-L1-FS, Picnic2-L1-FS) were considered but not evaluated as they do not pass the signature size requirement for DNS. It was found out that the evaluated algorithms perform signing and verification fast enough. However, two of them have significantly larger key sizes which make them less desirable. Nevertheless, XMSS and variants are not considered in any studies so far.
Kaliski [26] proposes to reduce the long signature size issue by applying a hash-based signature to only the KSK level, instead of the ZSK level. The integrity of the DNS records is protected by arranging them in the leaf nodes of the Merkle tree and involves verification of the root–synthesized KSK. In this case, the actual signature contains only an authentication path from a leaf to the root. The root node will be checked separately at the KSK level which requires hash-based signatures.

Chapter 4

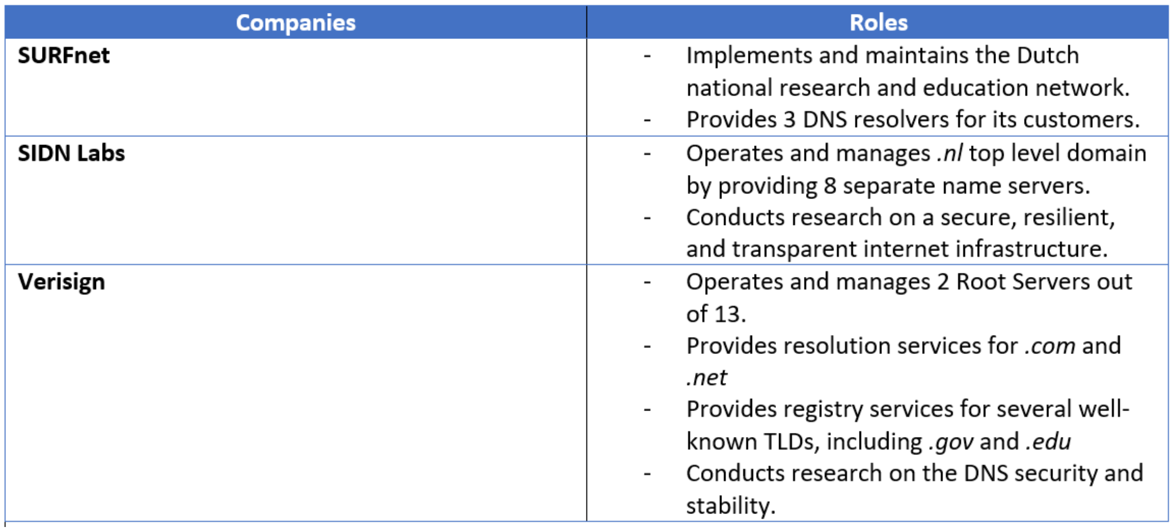
Post-Quantum DNSSEC Design and Variables

Redesigning the DNSSEC protocol from scratch is not usually the most efficient solution considering the time that it takes to build trust in the community, implement the prototype and deploy on large scale. Even though DNSSEC has been proposed more than a decade ago, it is still in the process of adoption by different operators [40]. Therefore, in this chapter, we will demonstrate and analyse the integration of the newly proposed approach to the existing protocol scheme (adjustment) from the signer and resolver perspective. Before beginning with the signer and resolver design, we gathered requirements through interviewing relevant parties such as an operator of a medium-sized TLD (SIDN), an operator of a large TLD and two root servers (Verisign) and a resolver operator (SURFnet). Table 4.1 summarizes the key responsibilities of each interviewed company. Requirements Analysis for signer and resolver can be found in Section 4.1.1 and Section 4.2.1 respectively. We later demonstrate the possible design flow for signer and resolver, taking into account current design choices such as update periods of zones. We identified important variables from the combination of requirements analysis and design flows and introduce them in Section 4.1.3 and 4.2.3 as signer and verifier variables. By this, we answer the first research question of identifying important variables to be traded off in the proposed approach (see Section 3.1).

4.1 Signer Design and Variables

4.1.1 Requirements

We conducted three video call interviews with Domain Name operators such as SIDN labs as a *.nl* TLD operator, SURFnet as a DNS resolver and Verisign as an operator for two of the Root Servers as well as ".com", ".net", ".gov" TLDs. Meetings



Companies	Roles
SURFnet	- Implements and maintains the Dutch national research and education network. - Provides 3 DNS resolvers for its customers.
SIDN Labs	- Operates and manages *.nl* top level domain by providing 8 separate name servers. - Conducts research on a secure, resilient, and transparent internet infrastructure.
Verisign	- Operates and manages 2 Root Servers out of 13. - Provides resolution services for *.com* and *.net* - Provides registry services for several well-known TLDs, including *.gov* and *.edu* - Conducts research on the DNS security and stability.

Table 4.1: Interviewed companies

were voice recorded for a later content referral.
In general, requirements can be categorized as general performance-related requirements and specific operator related requirements. Performance-related requirements include mainly fast resigning and zone updating. Below is a list of requirements for each operator.
Requirements by SIDN:

1. Complete signing of the zone: 10-11 mins
2. DNSSEC response size (majority of the responses) should not exceed 1,232 bytes [2].
3. Whole Zone Updating (including signing and checks) should not take more than 30 minutes.
4. Update Frequency analysis on the records should be done periodically if it is ever done.
5. Updates in the zone should be visible every 30 minutes.

Fast upfront signing and small signature size are a common performance-related requirement for TLDs.

Requirements by Verisign:

1. Ceremonial signing should occur every two months (as a Root DNS operator).
2. Upfront Tree signing - singing all the records before it gets (as a Root DNS operator), compare with dynamic signing.

3. ZSKs in the zone should be renewed latest every three months.

4.1.2 Design Flow

This section provides a detailed view of the integral functional flow of the offline signer, combined with the proposed approach, requirements, and current protocol standards. Note that it only covers offline signing, and online signing is kept out of the scope.

Upfront Signing Process. Most Root, TLD or authoritative name servers compute the signatures upfront. The signing flow is depicted in Figure 4.0 and each activity is explained below:

(A) To start with the signing process, the first step in the proposed approach will be the *analysis* of the frequency of changes for each signed Record type. It involves determining how many times each record set has been updated during a given period.

(B) Based on the identified range of update frequencies, those records that have similar update frequencies will be *grouped* in the same tree. This provides a high probability to keep the number of trees that potentially need to be updated low. For instance, having a tree that contains only one record with a high update frequency will force the number of nodes equal to the height of the tree to be updated. This will cause all signatures for the leaf records to be recomputed since at least one of the updated hash nodes is included in each signature. Figure 4.1 explains this correlation more detailed. In the figure, changes to the record and the effects on intermediary hash nodes are indicated in red. The change to the last leaf node affects the three hash nodes $a_{0,7}$, $a_{1,3}$ and $a_{2,1}$. The affected $a_{2,1}$ is also included in the signature of the first four records. Similarly, $a_{1,3}$ is part of a signature of a record y_4 and y_5 and node $a_{0,7}$ is part of a signature of a record y_7. It proves that a single record change forces to update each record signature. If there are N of this kind of frequently updated records, placing them separately over N trees will cause N different tree updates, thus, N new tree roots and even more signatures. In contrast, placing them in one tree would only cause one tree update at a time (note that the record changes are accumulated and applied at once every X duration determined by a zone operator, not every time a record change occurs). As one of the requirements mentioned in Section 4.1.1, analysis of update frequencies should be done *timely* since historical data on domain name changes can no longer hold as time passes. This is due to a couple of reasons one of which is a move of an inactive domain from one host to a different host where changes become much more frequent.

(C) After grouping the records by update frequencies, Merkle trees will be *constructed* from each group of records. The leaves of the tree are hash values of RRsets, and each parent node is the concatenation of the left and right child node.

(D) Merkle tree construction will initially require $2^{H+1} - 1$ computation of the hash nodes. Later, depending on the number of updates to the leaf nodes, the reconstruction of the tree may require the calculation of up to $2^{H+1} - 1$ hash noded in the worst-case scenario (where each leaf node has been updated simultaneously).

(E) The tree construction produces an integral root node that needs to be signed using the XMSS signature (G). As discussed in Approach Section 3.2, this ensures that the root node itself is trusted as it provides the authenticity and integrity of the record signatures.

(F) *Record signing* involves a binary search for indices of H_{mtree} number of hash nodes and outputs a signature that contains H_{mtree} number of concatenated hash nodes.

(G) *Signing* the root typically involves computing XMSS signature over the root of the Merkle tree.

(H) The activities such as *Signing* (G) the roots as well as *construction* (C) of Merkle tree are re-occurring processes. That is, any changes to the records need to be reflected in the root and record signatures. A new signature will be generated in two scenarios: either a tree has been updated or signatures are about to expire. Signatures belonging to the same tree will have the same expiration time.

4.1.3 Variables

The aforementioned Design flow reveals important variables to consider for a signer. In this section, we will identify and list relevant variables per flow activity (labelled with capital letters) which consequently answers the first research question - ”What are the important variables to be considered if we use Merkle tree Authentication and sign root of the tree with XMSS in DNSSEC?”. We also list important metrics to later analyse the impact of the subset of the variables on them as a part of the second Research Question covered in Section 6.

1. Update Frequency Analysis of Resource Records **(A)**.

   - Min and max update frequency in the zone: fr_{min}, fr_{max}.
     These variables indicate the highest and lowest update frequency among

the records during a period. For instance, in a certain period, the records that have never been updated set the min update frequency as zero in the zone.

2. Grouping Resource records with similar update frequencies **(B)**.

   - Frequency range difference variable (similarity level definer): fr_{diff}.
     This variable shows the maximum allowed difference of update frequencies of resource records in each tree. It implies the degree of similarity among resource records in each tree. For instance, $fr_{diff} = 20$ would mean that in any given tree, the difference between the highest and lowest update frequencies of records cannot be more than 20 times. The value of this variable should be determined after the update frequency analysis of zones and depends on the zone operator.

3. Merkle Tree Construction **(C,D,E,F)**.

   - Merkle tree Height: H_{mtree}. Height of the Merkle Tree is determined by the amount of the records each tree contains (number of leaves). It also determines the size of a record signature which needs to be within the limit of DNS response size requirement. This ensures to avoid a fragmentation issue in DNS (see 2.1).
   - Number of Trees: n_{mtree}.
     This variable shows how many trees with the determined fr_{diff} or H_{mtree} need to be constructed to cover all the signed Resource Records in the domain name operator.
   - Size of the Merkle Tree: $n_{records}$. This variable indicates the number of Records in a Merkle Tree.
   - Record Signature Size: ss. This variable indicates the size of the signature over an RRset for a specific domain.
   - Tree Generation Time: t_{mtree}. This variable indicates how much time it would require for a new tree to be constructed.

4. Root Signing **(G)**.
   Variables identified in this process mainly entails parameters relevant to XMSS algorithm. Since most of these parameters already have standard accepted values, we treat XMSS as a given; thus, the analysis and evaluation of the value choice of these variables will not be discussed further in this thesis.

   - Number of supported XMSS signatures: n_{sigs}.
     This variable shows how many roots (trees) can be signed in a zone. The

amount of leaf tree nodes in the XMSS tree correspond to this variable. A standardly accepted value can be used initially.

- XMSS tree Height: H_{xmss}. Similar to H_{mtree}, the height of the XMSS tree is closely related to the number of leaf nodes it contains.
- Root Signing Speed: t_{xmss}. This variable shows the time that it takes to sign the root of the tree. As a result of the signing process, an XMSS signature is generated. This variable will have a predicted value independent of the tree size and type.
- Winternitz security parameter: w. This variable shows how many bits of the message (in this case, root of the tree) need to be signed simultaneously.
- XMSS Signature Size: ss_{xmss}. This variable shows the size of the XMSS signature over the root of the tree.

5. Setting Zone Update time **(F)**.
   - Zone Update Period: T_{update}. This variable shows the period where the changes to the zones are processed. It is set based on the internal configuration of the zone operator. The analysis of this variable is out of the scope of this thesis.

Extended performance *metrics* for signer activities are listed below:

1. Single Record Signing speed: t_{rrsig}. This variable shows the time that it takes to generate the signature for the RRset in the tree. Note that this variable will have a different value depending on the tree, which will be discussed later.
2. Zone Signing Speed: $t_{zone-sig}$. This variable shows the time it takes to sign the entire zone upfront. Most of the record signatures are generated in advance and usually, the aggregated time of signing all records is more useful than the time spent on individual signature generation (t_{rrsig}).
3. Tree Update Time: t_{update}. This variable indicates the time that it takes for a tree to be updated. In the worst-case scenario where all the records have been changed, it is identical to tree generation time.

4.2 Resolver Design and Variables

4.2.1 Requirements

For resolver requirements, an interview was conducted with SURFnet. Fast verification of record signatures (verifying at least 1000 signatures per second [2]) is the only crucial requirement that was identified.

4.2.2 Design Flow

This section provides a detailed view of the integral functional responsibilities of the verifier as soon as the requested record is not cached, combined with the proposed approach and current protocol standards. When the resolver receives RRSIG records as a response, the very first task is to check the validity time of the signatures (A). It should be noted that all the records belonging to the same tree will have the same signature expiration date of the signature. If the signature is up-to-date, the verification process starts (C, D); otherwise, it is considered bogus. Figure 4.0 shows verification flow for the record verification. It omits the queries necessary to fetch records, signatures, and keys.
As mentioned in Merkle Tree 2.3.3 and proposed in Approach 3.2, a signature for a record contains $(r, authPath)$ where r indicates the index of the record in a tree and $authPath$ contains the required values of hashed nodes to verify the record. Verification involves traversing through the $authPath$ to the root node from a leaf node (record itself) and comparing the computed root node to an already trusted XMSS signed root node. If the root node (XMSS signature) has not been verified yet, then a standard XMSS signature verification process is followed up (E). Thus, a record is considered to be verified if traversing through authPath leads to a verified root node. The root signature will be cached (F) for a certain period depending on the zone requirement and will be updated periodically. This aims to reduce verification time as the XMSS signature verification process will not be repeated every time a new record signature from the same tree needs to be verified.

4.2.3 Variables

The aforementioned design flow reveals important variables to consider for a verifier. In this section, we will identify and list relevant variables as well as a metric per verifier flow activity that adds up to the list of variables for the first research question.

- Root Node Verification Speed: $t_{xmss-ver}$ This variable indicates the time to spend on verifying the root - XMSS signature.

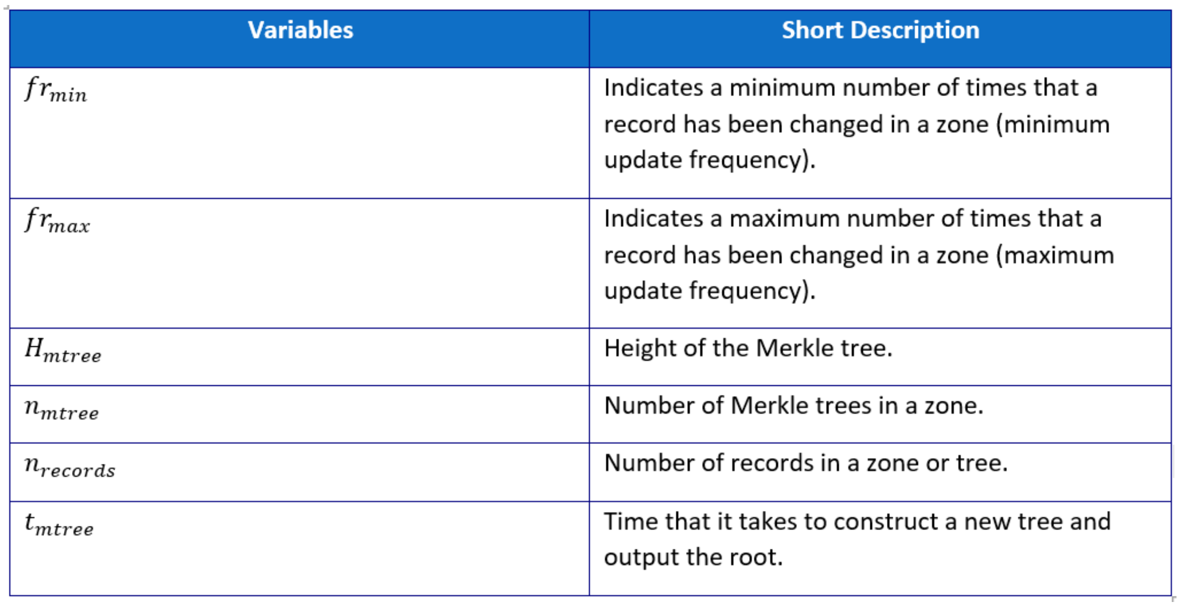
- Cache Update Period: $T_{caching}$. This variable indicates how long the verified XMSS signature should be cached.

For this research, we mark a single performance *metric* for the verifier to be analysed and evaluated further:

- Record Verification Speed: t_{ver}. This variable indicates the time spent on record verification by traversing through the hash nodes provided in the signature.

4.3 Summary and Conclusion

In this chapter, we shortly demonstrated signer and resolver perspectives by gathering requirements on their internal operations. Operator specific requirements concern mainly the size of the operator. For instance, Verisign's algorithm choice for signing the root of the trees in larger TLD zones would require supporting many more one-time signatures compared to that of smaller and medium-size zones. Thus, a different root signing algorithm (e.g. XMSSMT instead of XMSS) can meet the requirement by Verisign due to its support for large scale signature generation. Later, we tried to integrate the proposed approach to the standard signing and verification flow by taking some of the general requirements into account. It should be noted that no operator-specific such as "ceremonial signing" [41] by Verisign is reflected in the design as it requires additional modification that is out of the scope for this research. Furthermore, we also identified a list of variables from signer and resolver activities that can potentially have an impact on performance metrics - signature size, signing and verification speed. Table 4.2 and Table 4.3 summarizes important variables and metrics respectively.



Variables	Short Description
fr_{min}	Indicates a minimum number of times that a record has been changed in a zone (minimum update frequency).
fr_{max}	Indicates a maximum number of times that a record has been changed in a zone (maximum update frequency).
H_{mtree}	Height of the Merkle tree.
n_{mtree}	Number of Merkle trees in a zone.
$n_{records}$	Number of records in a zone or tree.
t_{mtree}	Time that it takes to construct a new tree and output the root.

Table 4.2: Important Variables

Metrics	Short Description
ss	Size of the signature over RRset.
t_{ver}	Time to spend on verifying the SS.
t_{rrsig}	Time to spend on signing a single record set (excluding root signing).
$t_{zone-sig}$	Time to spend on signing all the records in the zone.
t_{update}	Time to spend on updating a Merkle tree.

Table 4.3: Extended Metrics

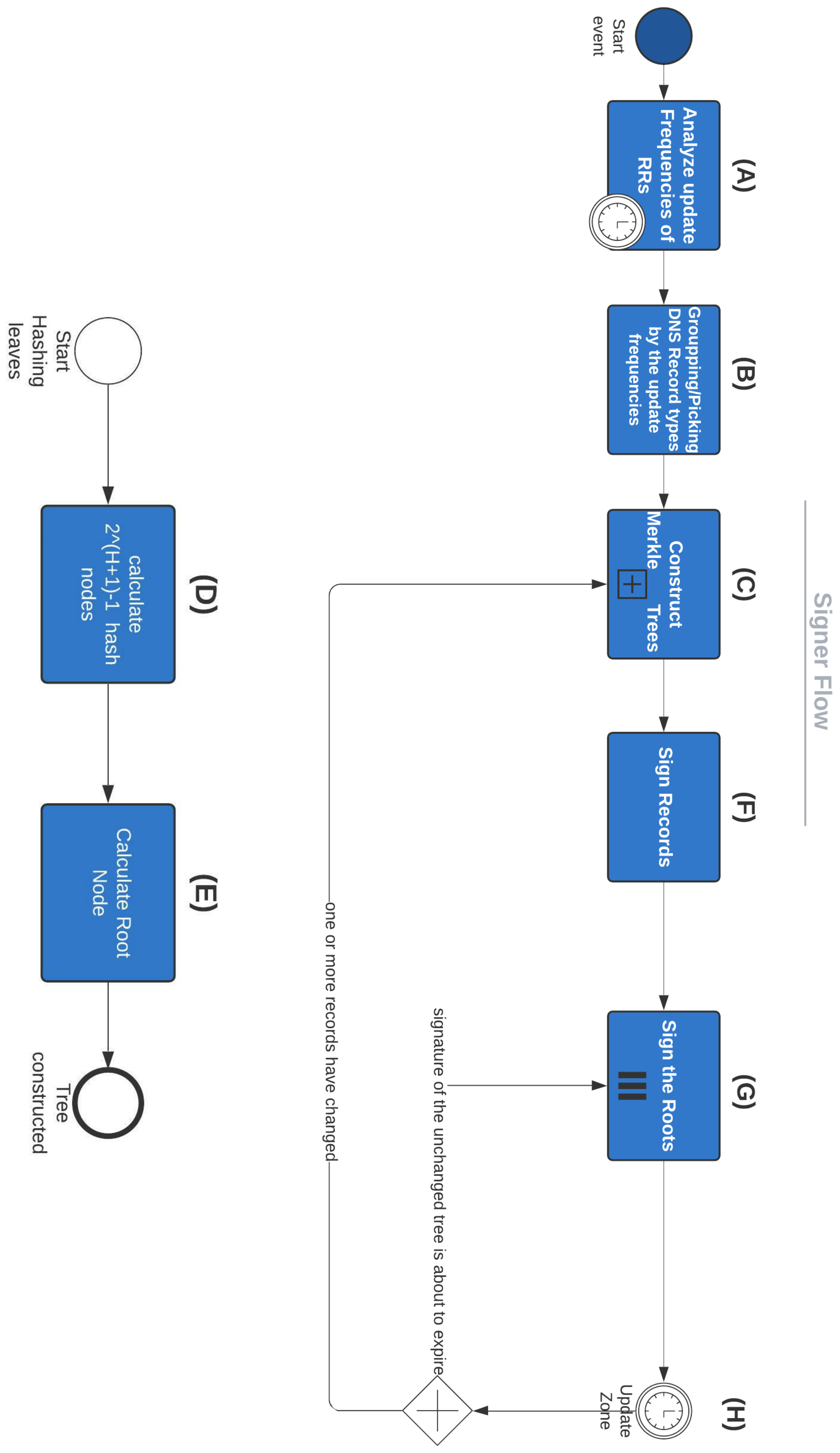

Signer Flow
Start event
(A)
Analyze update Frequencies of RRs
(B)
Groupping/Picking DNS Record types by the update frequencies
(C)
Construct Merkle Trees
(F)
Sign Records
(G)
Sign the Roots
(H)
Update Zone
signature of the unchanged tree is about to expire
one or more records have changed
Start Hashing leaves
(D)
calculate 2^(H+1)-1 hash nodes
(E)
Calculate Root Node
Tree constructed

Figure 4.0: Signer Flow

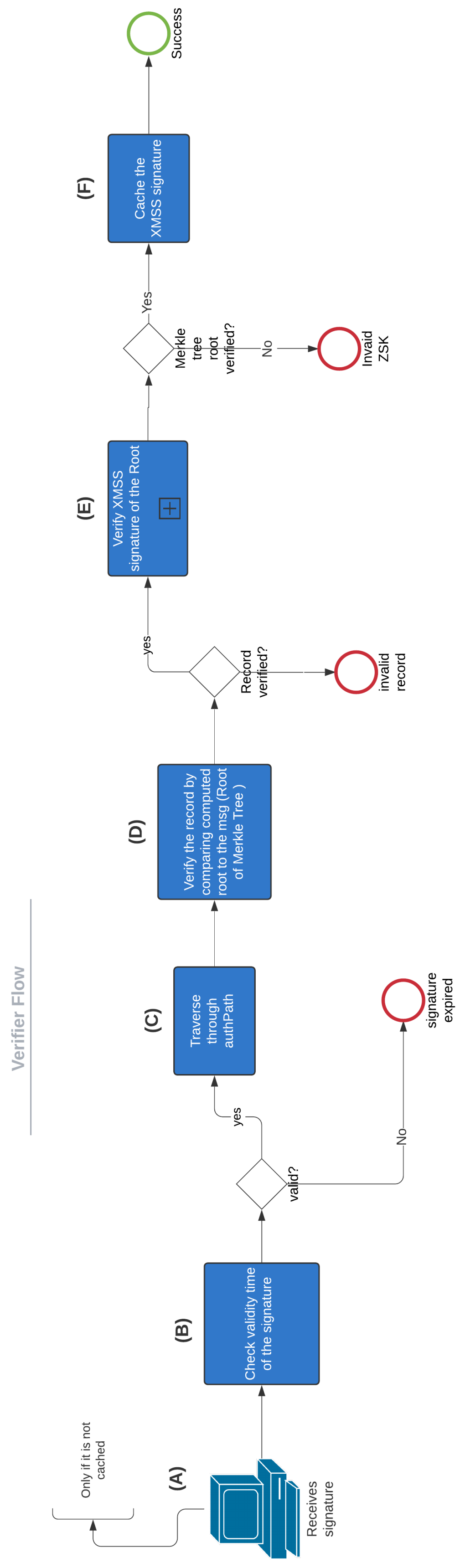

Verifier Flow
Only if it is not cached
(A)
Receives signature
(B)
Check validity time of the signature
valid?
No
signature expired
yes
(C)
Traverse through authPath
(D)
Verify the record by comparing computed root to the msg (Root of Merkle Tree)
Record verified?
invalid record
yes
(E)
Verify XMSS signature of the Root
Merkle tree root verified?
No
Invaid ZSK
Yes
(F)
Cache the XMSS signature
Success

Figure 4.0: Resolver Flow

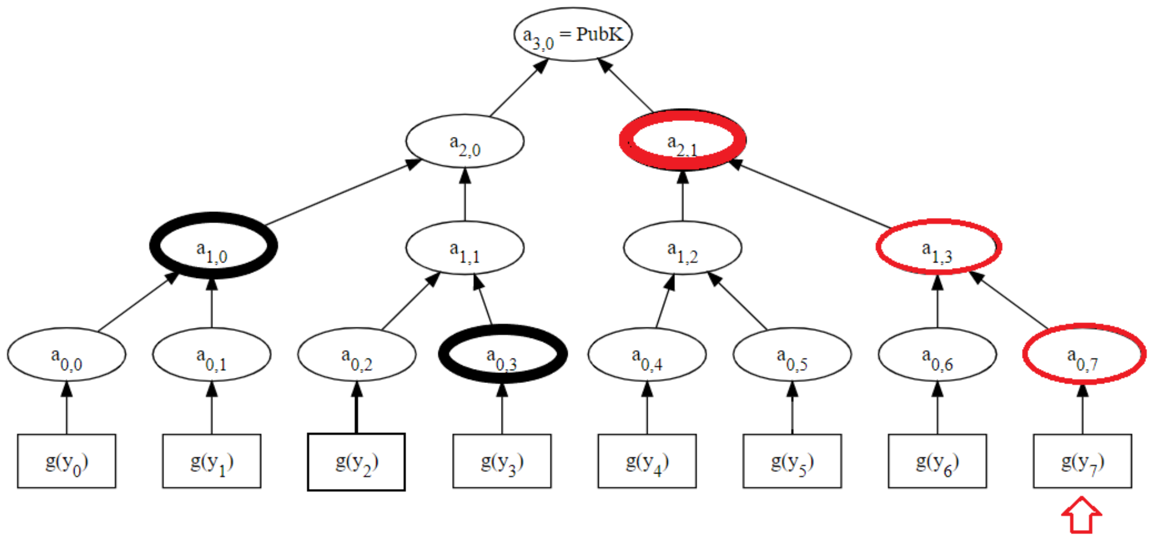


Figure 4.1: Single record change affecting other signatures.

Chapter 5

Methodology and Prototype

This chapter provides a guideline to find out the impact of the important variables on the metrics defined above (second research question) as well as explore an option of grouping domains based on update frequencies (third research question). The methodology section introduces the data sets that are used in the prototype and the procedures to analyse them. The prototype section further explains the technical sides of the signer and resolver implementation such as Merkle Tree generation, intermediate node calculation out of the generated tree (signature generation) and verification - traversing from provided index of leaf node to the root when a signature is provided.

5.1 Methodology

To practically evaluate the impact of the variables identified from the first research question and identify the upsides as well as downsides of the record grouping, we implemented a simple prototype for Signer 4.0 and Resolver flow 4.0 by mostly using real-life data for Merkle Tree construction. For the prototype, two types of the dataset were selected for the update frequency analysis of the signed records as shown in the signer flow. Below, we introduce the datasets that were used and gives details on the actual analysis that was carried out:

1. Root zone updates from git repository [42]. Since the root zone plays a significant role in the DNS hierarchy, we started the analysis process with a publicly available dataset repository for the root zone. This repository contains a history of updates to the Root zone between 2014-03-05 and 2021-02-12. For the root zone, DS, NSEC and SOA record types are usually signed and therefore, are expected to be the leaf nodes of the Merkle tree. Firstly, zone files were extracted from the repository which resulted in two zone files per day (5,522 zone files). In general, for the frequency update analysis of the domain

names, zone files in the past one year or even a month would be more relevant. Since TLD domains, particularly DS record changes are not expected to happen frequently, we decided to analyse the entire dataset to be able to distinguish frequent updates in records. The analysis includes three stages:

(a) parsing each zone file,

(b) in each zone file, grouping signed record sets by domain name per record type such as DS, NSEC or SOA records,

(c) comparing the contents of the record sets for each domain across the zone files and counting the differences per domain name for DS record changes.

2. Dataset on *.nl* Zone. Almost 60% of the domains are signed in the .nl zone. Considering the high adoption of DNSSEC, *.nl* is the most fitting medium-sized TLD zone that could give a good approximation on the performance metrics. The dataset was already pre-processed by SIDNlabs and spanned a whole year of DS record changes between 2020-12-21 and 2021-12-21. As opposed to the first dataset, it only contains domain names and counts of DS record changes per domain, not actual DS records themselves. Therefore, random SHA-256 hash values were generated as DS records for later use.

After the data analysis phase, prototype testing was carried out in two parts. First is *Single Tree* signing and verification performance testing depending on the variables. This part of testing aimed at smaller zones such as the Root zone where the amount of records and frequency of the record changes are a few times less compared to the *.nl* zone. However, considering a potential increase in the root zone in the future, some artificially generated DS records were added to the dataset and signing as well as verification with a single tree were simulated as well.
The second part was *Multi-Tree* signing and verification testing which aims at mainly mid or larger zones such as *.nl*. This testing has two sub-cases depending on the fixed or variable tree sizes. In the first sub-case, all trees contain equal number of records. All the DS RRsets in the zone were sorted by update frequencies and divided into fixed-size chunks where each chunk forms the leaves in a tree.
Second sub-case is having variable size of trees. That is, the RRset with only a very high update frequencies should be grouped in even smaller trees and the rest with much lower update frequencies can either be merged in one tree or can be divided into fixed-size chunks.
The simulation for most of the test cases was run 3-5 times. Since the standard deviation of the results was low, providing the average as final results suffices.

5.2 Prototype

The purpose of the prototype building is to create a proof of concept for the proposed approach and support theoretical analysis with practical results on the metrics. The analysis, implementation, and simulation were run on Anaconda Jupyter software installed in ASUS ZenBook UX360CA with 8GB Random Access Memory. The Following gives more insights into the three components of the prototype. The code for each component is provided in a git repository [43]. It should be noted that the efficiency of the algorithms used in prototype is not included in the objectives of this work. Hence, even though the technical choices were briefly mentioned, their implementation efficiency is out of the scope.

Tree Construction. To generate the Merkle Tree, an input parameter of a dictionary or set of leaf nodes is needed. In the case of the dictionary, the keys show domain names and values are corresponding RRset records of the type to be signed (in our case, DS records). Values are stored as leaf nodes in the tree after calculating their hashes and converting them into the digest. The rest of the implementation is trivial and involves concatenation and storing a hash of concatenated nodes in the tree. Note that hashed nodes are indexed from top to bottom, meaning that Root node has an index of 1 and Leaf nodes has index of maximum $2^{H_{mtree}+1}$ or $2 * n_{records}$. Important decisions in the tree implementation are dealing with the number of leaves that is not necessarily a power of two and whether or not to store all the intermediary nodes in the tree. For the former one, the approach of simply padding enough number of *null* nodes was used. Thus, only balanced trees were considered. For the latter, known as the tree traversal problem in literature, no special algorithm was taken into consideration thus all the nodes were saved in the tree. This means that a prototype might have a larger memory footprint. Tree construction ends with outputting the root node.

Signature Generation. As discussed in Approach 3.2 and Signer Design 4.1.2 sections, a signature over RRset contains the index information of the record (leaf node) and the $authPath$ which is a set of H_{mtree} number of hash nodes in the tree. Given the pre-constructed Merkle tree and index of the leaf node to be verified, the prototype outputs the intermediary nodes as $authPath$. Calculated $authPath$ together with the index of an RRset (leaf node) forms the record signature. Note that XMSS signature generation for the root node is excluded from the prototype since it has a predetermined effect on the signature size, signing and verification speed independent of the tree size. Thus, the prototype covers only signing the record without signing the 'synthesized ZSK'.

Signature Verification. To verify the record signature, the root of the tree needs to be known and trusted. This is normally achieved by verifying the XMSS signature over the root. Since signing the root has been skipped in the prototype, we trust the root node computed from the tree construction component. Additionally, the verification component takes input parameters of $\{$*index:leaf*$\}$ where a leaf is the plain record to be verified, depth of the tree that the leaf node belongs to and signature as an $authPath$ for that leaf node. Algorithm for the verification steps are as follows:

1. Hash the leaf node.
2. Add value from Step 1 to the queue as a dictionary pair $\{$*index:hash_value*$\}$.
3. Take the top *index:value* pair from the queue.
4. Check the index of the queue element from Step 3. If it is equal to the index of the root node, program compares the Root value to the value from Step 3 and returns $True$ or $False$. Otherwise, Step 4 followed.
5. Hash the concatenated value of Step 3 and the first hash node from the $authPath$.
6. Add *index:value* pair to the queue with where *index* is half of the previous index and *value* is from Step 5.
7. Repeat from Step 2 on.

The index value is needed for only optimization purposes. That is, if multiple records are to be verified, the Therefore, the verification process only entails traversing from the specified leaf node to the root through the $authPath$ provided by the signature and comparing the computed root node to the already trusted root. The prototype outputs $True$ or $False$ depending on the result of the comparison.

5.3 Summary

In the methodology section of this chapter, we provided motivations for dataset choices and briefly explained procedures for data analysis and prototype testing phases. The first phase, which is the *Single Tree* phase, was targeted at small zones; whereas, the *Multi-Tree* testing phase was mainly suitable for mid and large size zones. Later in the prototype section, we provided details on the components of the prototype such as Tree Construction, Signature Generation, and Signature Verification.

Chapter 6

Results and Evaluation

This chapter consists of two sections. The results section itself is divided into two categories: important outcomes obtained from *data analysis* and numerical results obtained from *prototype testing*. The evaluation section elaborates on the obtained results from prototype testing, provides correlation among the variables and the metrics and discusses trade-offs identified from prototyping. Overall, this chapter aims to provide answers to the second and third research questions.

6.1 Results

Data Analysis. Frequency analysis of the Root zone and *.nl* TLD zone indicate that the DS records for *.nl* domains tend to change much more frequently than in the Root zone. Table 6.1 provides some statistics over the data sets. As can be seen,

Variables	Root Zone	*.nl* TLD Zone
$n_{records}$(DS sets)	1372	4.084.519
fr_{max}	23	265
fr_{min} = 0	276	2.441.373

Table 6.1: Update Frequency Analysis for DS records.

Metrics	Variables
ss	H_{mtree} or $n_{records}$
t_{rrsig}	H_{mtree} or $n_{records}$
$t_{zone-sig}$	H_{mtree} or $n_{records}$, n_{mtree}, t_{update}
t_{ver}	H_{mtree} or $n_{records}$

Table 6.2: Variables that have direct impact on the Metrics

there are *1372* signed domains in the root zone and *4.084.519* in the *.nl* zone. In the root zone, the DS record that changed the most has been updated 23 times (fr_{max}

= 23) and belongs to the *pl.* domain. Meanwhile, the most frequently changed DS record in *.nl* zone changed 265 times in just a year. *276* domains in the root zone never updated their DS records over the past decade. Similarly, *2.441.373* DS records in *.nl* zone never get updated during a year.

We further analyse update frequency ranges for only *.nl* zone to identify records with similar update frequencies. For the root zone, this is not essential since the highest update frequency is low, and all records can easily be managed in one tree. Figure 6.1 shows the cumulative record distributions over the full frequency range ($0 \leq fr \leq 263$) for *.nl* zone. The majority of the records have update frequencies between $0 < fr < 50$. If we zoom in that range, Figure 6.2 shows that fr = 20 can be a good candidate for an upper bound for the records with low update frequencies. Thus, records with an update frequency higher than 20 are referred to as records with high update frequencies. Figure 6.3 demonstrates detailed statistics on records with high update frequencies. Update frequency ranges for this part of the records were chosen for only demonstration purposes. In total, *22* records have update frequencies higher than 20. Concerning records with low update frequencies ($fr < 20$), majority of the records are accumulated below $fr = 5$ (Figure 6.2). Therefore, we pick 5 to be the range determiner for records with low update frequencies ($5 < fr < 20$, $0 < fr < 5$ and $fr = 0$). Figure 6.4 shows the number of records with low update frequencies in each range.

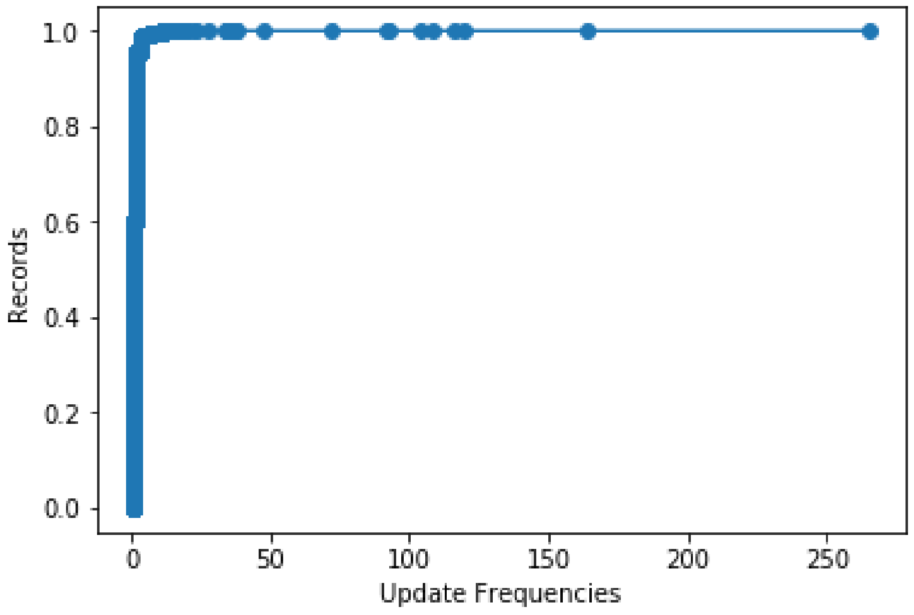


Figure 6.1: CDF over all data points.

Prototype Results. In this section, we provide numerical results for signature generation and verification speed affected by the previously identified variables. Since

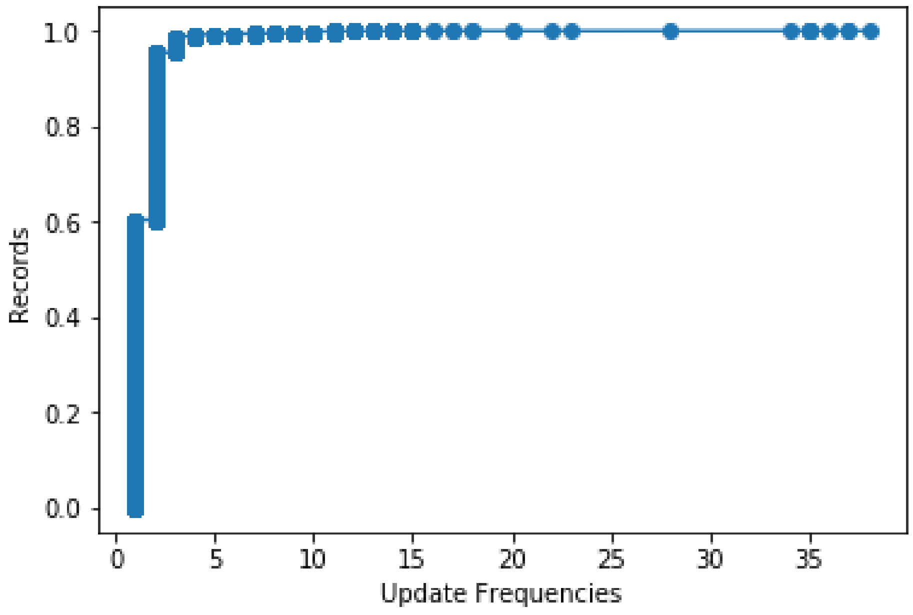


Figure 6.2: CDF over the data with frequencies $0 < fr < 50$.

we primarily focus on the record signing in this research, variables relating to XMSS signature generation (signing the root node) defined in Section 4.1.3 are excluded from testing. Remaining variables can have either direct or indirect impact on the metrics: *signature size*, *signing* (single record set and entire zone) and *verification speed*. Variables that have a direct impact are summarized in Table 6.2 and are the target for the impact evaluation. Indirect variables, on the other hand, play role in determining the direct variables, hence indirectly impacting the metrics. The reason why we mention the height of the tree (H_{mtree}) and number of records($n_{records}$) jointly in the table is as provided in Section 2.3.3, number of records in a tree precisely determines the height of the tree and can be written as $H_{mtree} = log_2(n_{records})$ so the impact of one of them on any metrics is relevant for the other one as well. The following provides results from each part of the testing of the direct variables on the metrics.

Single Tree Testing

As described in Methodology 5.2, for a small zone like the root zone, we constructed a single tree containing all DS RRsets from the root zone dataset and noted performance metrics. Table 6.3 below summarizes the results on the metrics dependent on the tree size (records in the zone). Note that artificially generated DS records (random hash values) were added to simulate a slightly larger Root zone with more records. Moreover, all the results refer to the worst-case scenarios. That is, zone

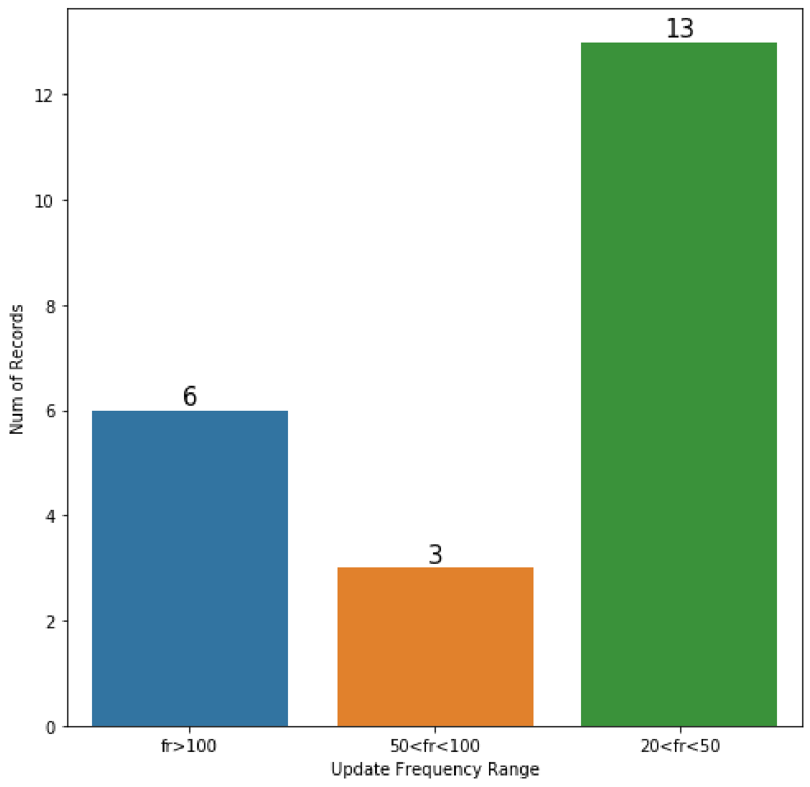


Figure 6.3: DS Records with High Update Frequencies.

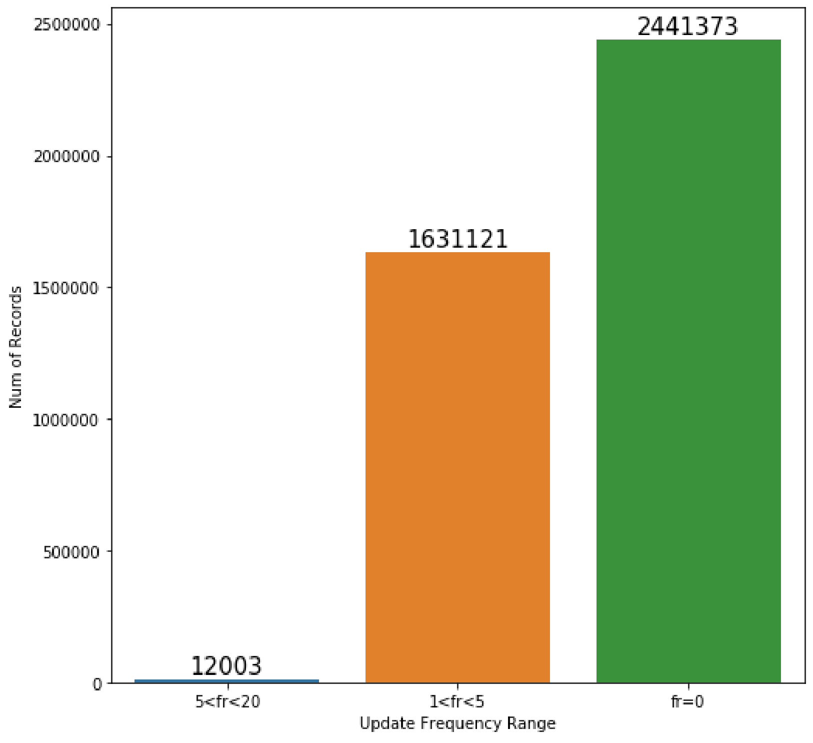


Figure 6.4: DS Records with Low Update Frequencies.

Variables	Results
$n_{records} = 2048$, $H_{mtree} = 11$	$t_{rrsig} \approx$ 0:00:00.002990 (one signature) $t_{zone-sig} \approx$ 0:00:04.84722 (2048 signatures) $t_{ver} \approx$ 0:00:00.00249 $t_{update} \lesssim$ 0:00:00.05559
$n_{records}$ =32,768, $H_{mtree} = 15$	$t_{rrsig} \approx$ 0:00:00.03989 $t_{zone-sig} \approx$ 0:08:16.9958 $t_{ver} \approx$ 0:00:00.00249 $t_{update} \lesssim$ 0:00:00.2474
$n_{records} = 1048576$, $H_{mtree} = 20$	$t_{rrsig} \approx$ 0:00:01.37902 $t_{zone-sig} \gg$ 1 day $t_{ver} \approx$ 0:00:00.00366 $t_{update} \lesssim$ 0:00:27.9664

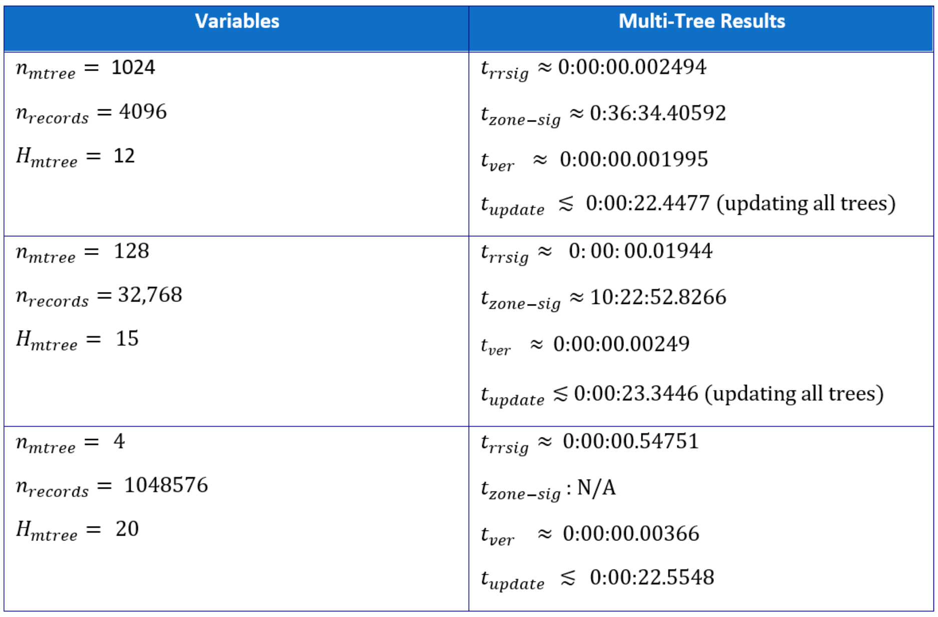
Table 6.3: Root Zone: Single Tree Performance in relation with the Tree Size.

signing time ($t_{zone-sig}$) includes resigning all of the DS records which is true under the condition when all the signatures are expired at the same time or at least one record gets updated within the zone update period. t_{update} also shows the upper bound for the time that it takes to update the entire tree in case all the leaf nodes have been changed.

Multi-Tree Testing

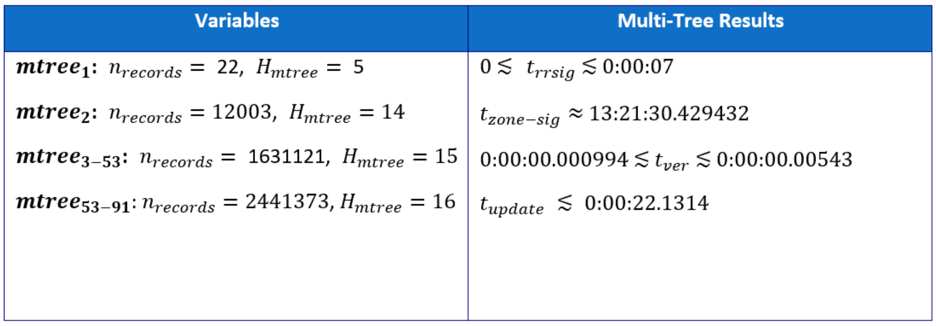
Multi-Tree testing aims at medium and larger zones such as *.nl* with higher frequency changes. Depending on the Tree Size of each tree, we divided this part into two sub-parts:

1. Fixed-Size Trees. In this part of the testing, records in the zones were sorted by update frequencies and divided into the same size trees. As shown in Table 6.4, with 4.084.519 DS records in *.nl* zone, the possible grouping would be 1024 small trees each containing 4096 RRsets, 128 large trees with 32,768 or even bigger and fewer trees. In this part of the testing, final t_{rrsig} shows the average of t_{rrsig} for records from different trees. t_{update} shows the update time for the entire zone. The main remark from the result is full record signing takes much less time with many smaller trees than a few bigger trees (shown as not available in the table 6.4 due to much longer execution time). Moreover, simultaneously updating all records in the zone are an equally costly operation regardless of the tree sizes.



Variables	Multi-Tree Results
$n_{mtree} = 1024$ $n_{records} = 4096$ $H_{mtree} = 12$	$t_{rrsig} \approx$ 0:00:00.002494 $t_{zone-sig} \approx$ 0:36:34.40592 $t_{ver} \approx$ 0:00:00.001995 $t_{update} \lesssim$ 0:00:22.4477 (updating all trees)
$n_{mtree} = 128$ $n_{records} = 32{,}768$ $H_{mtree} = 15$	$t_{rrsig} \approx$ 0: 00: 00.01944 $t_{zone-sig} \approx$ 10:22:52.8266 $t_{ver} \approx$ 0:00:00.00249 $t_{update} \lesssim$ 0:00:23.3446 (updating all trees)
$n_{mtree} = 4$ $n_{records} = 1048576$ $H_{mtree} = 20$	$t_{rrsig} \approx$ 0:00:00.54751 $t_{zone-sig}$: N/A $t_{ver} \approx$ 0:00:00.00366 $t_{update} \lesssim$ 0:00:22.5548

Table 6.4: Fixed Size Multi-Tree Variations and Results.



Variables	Multi-Tree Results
$\boldsymbol{mtree_1}$: $n_{records} = 22$, $H_{mtree} = 5$ $\boldsymbol{mtree_2}$: $n_{records} = 12003$, $H_{mtree} = 14$ $\boldsymbol{mtree_{3-53}}$: $n_{records} = 1631121$, $H_{mtree} = 15$ $\boldsymbol{mtree_{53-91}}$: $n_{records} = 2441373$, $H_{mtree} = 16$	$0 \lesssim t_{rrsig} \lesssim$ 0:00:07 $t_{zone-sig} \approx$ 13:21:30.429432 0:00:00.000994 $\lesssim t_{ver} \lesssim$ 0:00:00.00543 $t_{update} \lesssim$ 0:00:22.1314

Table 6.5: Variable Size Multi-Tree Results.

2. Variable Size Trees. In Variable Size Trees, all the records with the high update frequency ($fr > 20$) were grouped in one tree. RRsets with low update frequencies ($fr < 20$) are spread over three subcategories corresponding to Figure 6.4. Frequencies within range 5 to 20 are grouped in one tree with H_{mtree} = 14. RRsets with frequency updates within range 1 to 5 are grouped into 50 trees each having H_{mtree} = 15. Finally, the non-changed RRsets are grouped into 38 larger trees each with H_{mtree} = 16. Table 6.5 summarizes these results. The minimum and maximum value range for metrics depend on the size of the tree a record belongs to.

6.2 Evaluation

6.2.1 Impacts of variables on the Metrics

Results from single-tree testing indicate that a single tree for Root Zone would be more than enough to sign DS records and potentially, other types of records such as NSEC and SOA can be merged into the same tree considering the amount of the leaves will not exceed 32,768 ($H_{mtree} = 15$) since the time to spend on zone-signing increases drastically. On the other hand, time spent on signature verification increases only slightly as the height of the tree increases. Thus, the impact of the direct variables (particularly, H_{mtree}) on the metrics can be summarized as follows:

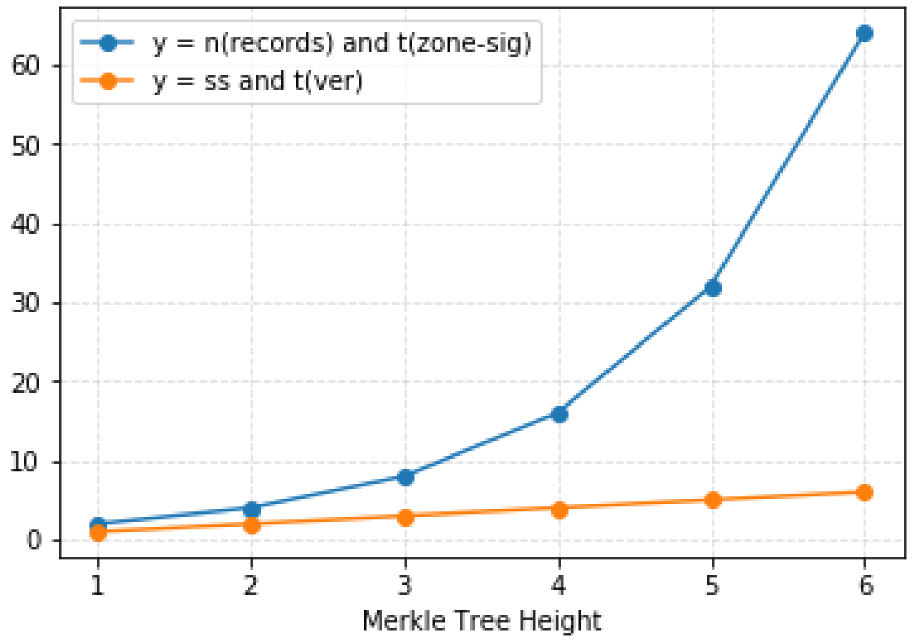


Figure 6.5: Correlation of H_{mtree} with $n_{records}$ and metrics

1. H_{mtree} on ss *(signature size)*. As explained in Section 2.3.3, Merkle Tree $authPath$ contains exactly H number of intermediary hash nodes to be traversed to the root. Since signature for the RRset contains $authPath$ along with the index of the RRset, the impact of tree height can be shown as linearly $ss \sim H_{mtree}$. Since H_{mtree} has direct correlation with $n_{records}$, the impact chain can be rewritten as $ss \sim H_{mtree} \sim log_2(n_{records})$.

2. H_{mtree} on t_{rrsig}. As described in Prototype 5.2, signature generation outputs H number of hash nodes. This involves binary search over all the $2^{H+1} - 1$ hash nodes to find the correct neighboring nodes as depicted in 2.4. Therefore, time complexity can be written as an exponential dependence on the height of the

tree $t_{rrsig} \sim 2^{H_{mtree}+1}$. Rewriting this would yield $t_{rrsig} \sim 2^{H_{mtree}+1} \sim n_{records}$. Table 6.3 and 6.4 also prove the correlation.

3. H_{mtree}, n_{mtree} and t_{update} on $t_{zone-sig}$. The metric $t_{zone-sig}$ is an extended version of t_{rrsig} where it includes updating existing trees and signing not a single RRset in a given tree but all RRsets in the zone. Separate evaluation of the Aforementioned three variables have different impacts on this metrics and need to be evaluated separately. Additionally, certain conditions should also be imposed on the variables to prove the impact analysis valid.

   (a) H_{mtree}. Keeping the total number of records in the zone stable, time to spend on signing the entire zone is exponentially dependent on the H_{mtree}. Refer to table 6.4 for comparison of $t_{zone-sig}$ values.

   (b) n_{mtree}. Only in the multi-tree zone, keeping the H_{mtree} in each tree constant, increase in the amount of trees (new records in the zone) require more time to sign the entire zone.

   (c) t_{update}. Time to spend on updating the tree is exponentially dependent on the height of the tree. Thus, it has linear impact on the $t_{zone-sig}$.

4. H_{mtree} on t_{ver}. As depicted in Resolver flow 4.0 and explained in 5.2, verification requires re-construction of the tree by using the provided H_{mtree} number of hash nodes in signature. This yields the same correlation as the signature size $t_{ver} \sim ss \sim H_{mtree}$. Table 6.3 and 6.4 demonstrate a small increase level in verification time.

Figure 6.5 summarizes the impact of tree height on important metrics.

6.2.2 Trade-off in Multi-Tree Zones

The downside of the fixed-size trees is that if the number of records that change frequently (more than 20 times in a year) is rather small - only 22 RRsets in case of *.nl* zone (see Figure 6.3) then having 2048 leaves in a single tree forces to include RRsets from a less frequency category. Since an update in the tree causes resigning of all the records in the tree, frequently changed RRset records cause the resigning of the other leaves in the same tree which is an expensive operation as proved above. On the other hand, the disadvantage of having smaller trees with less than 2048 leaves is that having many small trees mean many tree roots to be signed using XMSS. In case all the roots are concatenated and signed as a single root (DNSKEY record), this could exceed the upper limit for standard XMSS signature and cause fragmentation in the DNS packet. On the other hand, if the tree roots are signed separately as a new record type, this may require relatively more one-time

XMSS keys (for larger zones, even more). Variable Size Trees can be a common ground between many small trees and the high signature generation time of large trees. With variable size trees, RRsets with the highest frequency updates can be grouped in a small tree and the rest of the RRsets with low update frequencies can be grouped in fixed medium-size or large-size trees depending on the available resources. Even if the size of the tree is large with leaves being RRsets with low update frequencies, the signature generation of those records might need to happen only a few times in a year. Even though the record signatures have been expired, if there was no change in the RRsets, record signatures will have the same value except the root node will need to be resigned.

Chapter 7

Conclusions and Discussion

The conclusion section in this chapter provides the main takeaways from this research. The discussion section gives an insight into the limitations of this thesis and recommendations for future work.

7.1 Conclusions

The goal of this research was to experiment with the new two-layer approach for quantum-proof DNSSEC and perform variable correlation as well as trade-off analysis. We began with identifying variables from signer and resolver design flows. In total, 16 variables concerning signers and three variables concerning resolvers were listed in Section 4. The importance degree of the variables was mainly determined based on their direct impact on the metrics (signature size, signature generation and verification speed for RRsets). Thus, Merkle Tree Height, a total number of records as well as trees are classified as ”important” or variables with a direct impact on the metrics. It should be noted that although we identified variables relating to the root signing with XMSS, for this research, further testing and analysis of the XMSS related variables were skipped.
To analyse the impact of the variables identified from the first research question on the three performance metrics, the prototype was developed, and testing has been done. The key takeaways from this research are listed below:

1. Signature Size solely depends on the height of the Merkle Tree. Although larger trees (we define large trees as $H_{mtree} > 15$) lead to larger signatures and a drastic increase in total signature generation time of all the records in the tree, *signature verification* is the least impacted.

2. Updating the entire tree is not a costly operation per se. However, regenerating signatures for all the leaves in the tree is a costly operation. Since a single record update in the tree modifies at least one node in every signature

(*authPath*), all the record signatures are affected by a single update in the tree. Therefore, updates are inefficient in larger trees causing many signature updates.

3. Smaller zones up to 32,768 records do not necessarily need to split the records into smaller trees as the entire tree signing is faster enough to meet the requirement even in the case of frequent tree updates (see table 6.3). On the other hand, medium and large-sized zones should strive to reduce the number of signature updates affected by record changes and improve the signing speed.

4. Keeping the total number of records the same in the zone, grouping records into large size trees utilize high memory consumption compared to grouping them into small size trees.

5. Grouping Domains into variable size trees by update frequencies can reduce costly signature generation process to only hundreds of records that require a couple of minutes to sign.

7.2 Discussion

7.2.1 Limitations and Remarks

Below we acknowledge and list the existing limitations of this research. Firstly, due to limited hardware memory resources, the programming language choice (Python) and time constraints, the tests for larger trees ($H_{mtree} = 15$) could not be repeated and executed only once. Although initially all the signatures were also planned to be stored, due to the above-mentioned limitations, it could not be done. All the provided results on metrics aimed at displaying the relative impact of different parameters, not precise numbers for metrics.
Secondly, this research mainly focuses on the record signing by the possibility of grouping the records. As demonstrated in Approach 3.2 and Design 4.1, ZSK signing in current design corresponds to root signing using XMSS in the proposed approach. Variable analysis of XMSS signature such as winternitz parameter as well as XMSS algorithm variations are not studied in this thesis since there are already plenty of studies regarding analysis of optimal parameters [44] and improving verification speed. Additionally, an interactive platform has also been provided by Bas Westerbaan [45] to check the impact of different XMSS parameters on signature size, the number of supported signatures and the number of hashes required for verification.

7.2.2 Future Work

Based on the limitations acknowledged in this research, certain suggestions can be made to improve the results and complete the study. Firstly, the implementation including tree traversal and signing process can be optimised by using C or Rust programming languages. Secondly, hardware resources would help approximate the results much more accurately and most importantly, store the pre-computed signatures. It includes repeating the test cases 10s of times for larger trees, storing the signatures or pre-computed hash nodes. Much larger and dynamically signed zones such as *.com* may require special modification to the design and algorithm choice (e.g $XMSS_{MT}$ instead of $XMSS$ for root signing). This needs to be studied separately. Finally, a real-world test should be done with the optimised implementation.

Bibliography

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997. [Online]. Available: https://doi.org/10.1137/S0097539795293172

[2] M. Mueller, J. de Jong, M. van Heesch, B. Overeinder, and R. van Rijswijk - Deij, "Retrofitting post-quantum cryptography in internet protocols: A case study of dnssec," *Computer communication review*, vol. 50, no. 4, pp. 49–57, Oct. 2020.

[3] "Tld dnssec report," http://stats.research.icann.org/dns/tld_report/, Aug 2021.

[4] "What is dns cache poisoning? — dns spoofing," https://www.cloudflare.com/learning/dns/dns-cache-poisoning/.

[5] J. da Silva Damas, M. Graff, and P. A. Vixie, "Extension Mechanisms for DNS (EDNS(0))," RFC 6891, Apr. 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc6891.txt

[6] A. Herzberg and H. Shulman, "Fragmentation considered poisonous, or: One-domain-to-rule-them-all.org," 10 2013.

[7] "How dnssec works," https://www.cloudflare.com/dns/dnssec/how-dnssec-works/.

[8] R. van Rijswijk-Deij, M. Jonker, and A. Sperotto, "On the adoption of the elliptic curve digital signature algorithm (ecdsa) in dnssec," in *Proceedings of the 12th Conference on International Conference on Network and Service Management*, ser. CNSM 2016. Laxenburg, AUT: International Federation for Information Processing, 2016, p. 258–262.

[9] N. Research, "Dnssec fuels new wave of dns amplification," https://blog.nexusguard.com/dnssec-fuels-new-wave-of-dns-amplification.

[10] R. van Rijswijk-Deij, K. Hageman, A. Sperotto, and A. Pras, "The performance impact of elliptic curve cryptography on dnssec validation," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 738–750, 2017.

[11] O. Surý and R. Edmonds, “EdDSA for DNSSEC,” Internet Engineering Task Force, Internet-Draft draft-ietf-curdle-dnskey-eddsa-00, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-curdle-dnskey-eddsa-00

[12] J. Rijneveld, “Practical post-quantum cryptography,” https://joostrijneveld.nl/thesis/.

[13] J. Buchmann, E. Dahmen, and M. Szydlo, *Hash-based Digital Signature Schemes*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 35–93. [Online]. Available: https://doi.org/10.1007/978-3-540-88702-7_3

[14] B. Preneel, *Hash Functions*. Boston, MA: Springer US, 2011, pp. 543–553. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5_580

[15] B. Kapoor and P. Pandya, “Chapter 2 - data encryption,” in *Cyber Security and IT Infrastructure Protection*, J. R. Vacca, Ed. Boston: Syngress, 2014, pp. 29–73. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124166813000021

[16] L. Lamport, “Constructing digital signatures from a one way function,” Tech. Rep. CSL-98, October 1979, this paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010. [Online]. Available: https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/

[17] G. Becker and R. universität Bochum, “Merkle signature schemes, merkle trees and their cryptanalysis.”

[18] A. Hülsing, “W-ots+ – shorter signatures for hash-based signature schemes,” in *Progress in Cryptology – AFRICACRYPT 2013*, A. Youssef, A. Nitaj, and A. E. Hassanien, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 173–188.

[19] R. C. Merkle, “Secrecy, authentication, and public key systems.” Ph.D. dissertation, Stanford, CA, USA, 1979, aAI8001972.

[20] M. Szydlo, “Merkle tree traversal in log space and time,” vol. 3027, 05 2004, pp. 541–554.

[21] P. Berman, M. Karpinski, and Y. Nekrich, “Optimal trade-off for merkle tree traversal,” *Theoretical Computer Science*, vol. 372, no. 1, pp. 26–36, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397506008693

[22] M. Jakobsson, T. Leighton, S. Micali, and M. Szydlo, “Fractal merkle tree representation and traversal,” in *Topics in Cryptology — CT-RSA 2003*, M. Joye, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 314–326.

[23] A. Hülsing, S. Gazdag, D. Butin, and J. Buchmann, “Hash-based signatures : An outline for a new standard,” 2014.

[24] J. Buchmann, E. Dahmen, and A. Hülsing, “Xmss – a practical forward secure signature scheme based on minimal security assumptions,” in *IN PROCEEDINGS OF THE 4TH INTERNATIONAL CONFERENCE ON POST-QUANTUM CRYPTOGRAPHY, PQCRYPTO’11*. Springer, 2011, pp. 117–129.

[25] M. Bellare and P. Rogaway, “Collision-resistant hashing: Towards making uowhfs practical,” in *Advances in Cryptology — CRYPTO ’97*, B. S. Kaliski, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 470–484.

[26] B. Kaliski, “Securing the dns in a post-quantum world: Hash-based signatures and synthesized zone signing keys,” https://circleid.com/posts/20210122-securing-the-dns-in-a-post-quantum-world-hash-based-signatures/, Jan 2021.

[27] S. Micali, “Efficient certificate revocation,” 07 2000.

[28] A. Perrig, R. Canetti, J. Tygar, and D. Song, “The tesla broadcast authentication protocol,” *RSA CryptoBytes*, vol. 5, 11 2002.

[29] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine, “Authentic third-party data publication,” vol. 73, 01 2000, pp. 101–112.

[30] S. Micali, M. Rabin, and J. Kilian, “Zero-knowledge sets,” in *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, 2003, pp. 80–91.

[31] R. Rivest and A. Shamir, “Payword and micromint: Two simple micropayment schemes,” in *CryptoBytes*, pp. 69–87.

[32] “How log proofs work - certificate transparency,” https://sites.google.com/site/certificatetransparency/log-proofs-work.

[33] D. Butin, J. Wälde, and J. Buchmann, “Post-quantum authentication in openssl with hash-based signatures,” in *2017 Tenth International Conference on Mobile Computing and Ubiquitous Network (ICMU)*, 2017, pp. 1–6.

[34] D. Sikeridis, P. Kampanakis, and M. Devetsikiotis, "Post-quantum authentication in tls 1.3: A performance study," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 71, 2020.

[35] P. Kampanakis, P. Panburana, E. Daw, and D. V. Geest, "The viability of post-quantum x.509 certificates," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 63, 2018.

[36] S. Suhail, R. Hussain, A. Khan, and C. S. Hong, "On the role of hash-based signatures in quantum-safe internet of things: Current solutions and future directions," *IEEE Internet of Things Journal*, vol. 8, no. 1, pp. 1–17, 2021.

[37] R. van Rijswijk-Deij, A. Sperotto, and A. Pras, "Making the case for elliptic curves in dnssec," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, p. 13–19, Sep. 2015. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/2831347.2831350

[38] S. Gourley and H. Tewari, "Blockchain backed dnssec," 07 2018.

[39] D. Moody, G. Alagic, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and J. Alperin-Sheriff, "Status report on the second round of the nist post-quantum cryptography standardization process," 2020-07-22 2020.

[40] N. Sullivan, "Dnssec: An introduction," Aug 2020. [Online]. Available: https://blog.cloudflare.com/dnssec-an-introduction/

[41] "The dnssec root signing ceremony." [Online]. Available: https://www.cloudflare.com/en-gb/dns/dnssec/root-signing-ceremony/

[42] T. Finch, "Saveroot: Archive of the dns root zone." [Online]. Available: https://github.com/fanf2/saveroot

[43] S. Jafarli, "Merkle-tree-for-dnssec: This repository is part of the simulation for my thesis dedicated to securing dns in post-quantum era." [Online]. Available: https://github.com/sjafarli/merkle-tree-for-dnssec.git

[44] A. Hülsing, L. Rausch, and J. Buchmann, "Optimal Parameters for XMSSMT," in *1st Cross-Domain Conference and Workshop on Availability, Reliability, and Security in Information Systems (CD-ARES)*, ser. Security Engineering and Intelligence Informatics, A. Cuzzocrea, C. Kittl, D. E. Simos, E. Weippl, and L. Xu, Eds., vol. LNCS-8128. Regensburg, Germany: Springer, Sep. 2013, pp. 194–208, part 2: Security Engineering. [Online]. Available: https://hal.inria.fr/hal-01506577

[45] B. Westerbaan, “Interactive dashboard for xmss parameters and algorithm variations.” [Online]. Available: https://westerbaan.name/~bas/hashcalc/