VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KNIHOVNA A SADA NÁSTROJŮ PRO POUŽITÍ DNSSEC NA SERVERU

LIBRARY AND TOOLS FOR SERVER-SIDE DNSSEC IMPLEMENTATION

DIPLOMOVÁ PRÁCE
MASTER’S THESIS

AUTOR PRÁCE — Bc. JAN VČELÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR MATOUŠEK, Ph.D.
SUPERVISOR

BRNO 2014

Abstrakt

Tato práce se zabývá analýzou současných open source řešení pro zabezpečení DNS zón pomocí technologie DNSSEC. Na základě provedené rešerše je navržena a implementována nová knihovna pro použití na autoritativních DNS serverech. Cílem knihovny je zachovat výhody stávajících řešení a vyřešit jejich nedostatky. Součástí návrhu je i sada nástrojů pro správu politiky a klíčů. Funkčnost vytvořené knihovny je ukázána na jejím použití v serveru Knot DNS.

Abstract

This thesis deals with currently available open-source solutions for securing DNS zones using the DNSSEC mechanism. Based on the findings, a new DNSSEC library for an authoritative name server is designed and implemented. The aim of the library is to keep the benefits of existing solutions and to eliminate their drawbacks. Also a set of utilities to manage keys and signing policy is proposed. The functionality of the library is demonstrated by it's use in the Knot DNS server.

Klíčová slova

DNS, DNSSEC, správa klíčů, politika podepisování, Knot DNS, OpenDNSSEC, BIND, NSD, PowerDNS

Keywords

DNS, DNSSEC, key management, signing policy, Knot DNS, OpenDNSSEC, BIND, NSD, PowerDNS

Citace

Jan Včelák: Library and Tools for Server-Side DNSSEC Implementation, diplomová práce, Brno, FIT VUT v Brně, 2014

Library and Tools for Server-Side DNSSEC Implementation

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Petra Matouška, Ph.D., technické konzultace probíhaly se zaměstnanci sdružení CZ.NIC. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Jan Včelák
27. května 2014

Poděkování

Děkuji všem, kteří se mnou měli trpělivost při psaní této práce. Také děkuji kolegům ze sdružení CZ.NIC, kteří mi ochotně odpovídali na dotazy z praxe, pomohli mi rozhodnout navrhované alternativy nebo jakkoli přispěli svými nápady a připomínkami k výsledné podobě této práce.

Contents

Chapter 1

Introduction

The Domain Name System (DNS) is a critical part and one of the fundamental building blocks of the Internet. It is used not only for address resolution, but it is essential for e-mail delivery, network service location, and it can store almost any kind of data in general. Currently, almost every communication in the Internet starts with a DNS query.

The DNS system was not designed to provide strong security. It contains only a simple protection against various packet interception attacks, which still allows an attacker to modify the response message transmitted from a server to a client. To prevent these kind of attacks, the Domain Name System Security Extensions (DNSSEC) mechanism was proposed and implemented.

DNSSEC uses public-key cryptography to ensure the integrity and authenticity of the data in the answer messages. It has been designed and deployed gradually due to various problems identified at the time of the technology adaptation. As a result, a lot of existing DNSSEC tools have accustomed to certain mechanisms, which are suboptimal and have been preserved mainly for historical reasons.

At the moment, there are various open-source libraries oriented on DNSSEC validation. But there is no one to provide server-side DNSSEC management. The aim of this thesis is to design and implement an open-source DNSSEC library, which will cover the functionality necessary for securing DNS zones using the DNSSEC technology. The library is intended to be used by authoritative name servers, zone signing utilities, and other software for management of DNSSEC at the zone operator side. Great emphasis will be put on eliminating the problems of existing solutions.

At the beginning of this thesis, the DNS and DNSSEC are described. The explanation of the DNS is limited to the features required for understanding of DNSSEC mechanisms, that is the architecture of the DNS and the communication protocol. The DNSSEC description includes the motivation for the DNSSEC deployment, analysis of threats the technology is able to prevent, and how the public-key cryptography is applied to achieve these goals. The information is primarily focused on the aspects relevant for authoritative name servers.

Then, in order to define the requirements for the new library, existing open-source DNSSEC solutions with a similar specialization are analyzed and their advantages and drawbacks are identified. Based on the findings and use cases for the library, the new library is designed. Also a set of utilities to be distributed with the library is proposed.

Finally, the DNSSEC library is implemented and it's functionality demonstrated with the Knot DNS server.

Chapter 2

Domain Name System

This chapter describes the principles of the *Domain Name System* (DNS), which are necessary for understanding of DNSSEC mechanisms. The description covers the DNS data model, the architecture of the system, and the protocol operation.

In early 1980s, TCP/IP protocol suite was developed and became the standard for communication on an experimental wide-area computer network ARPANET, predecessor of the Internet. When the TCP/IP protocol was implemented in Berkeley's BSD Unix operating system, the connection to ARPANET became possible for a lot more organizations. The number of hosts within the network started to grow rapidly [1].

The IP protocol uses numeric IP addresses to identify hosts in the network. As numeric addresses are difficult to remember, hosts were assigned names and name-to-address mappings were put into a single file `HOSTS.TXT`. The file was maintained by *Network Information Center* (NIC) and every member of the network could fetch an up-to-date version of this file from a well-known FTP server [27, 28]. The process of retrieving of an IP address from a hostname is called *address resolution*. This approach to the resolution process is still available on most Unix operating systems, where the mappings are usually set in the `/etc/hosts` file.

The downside of a single file with name-to-address mappings is a bad scalability. A copy of the file has to be present on every host, the lookup performance gets worse with more hosts, and there is no mechanism to notify the hosts about the source file modification. In addition, the hostnames must be unique and there was no authority, which could work out arisen conflicts. The DNS delivered a solution for these problems.

2.1 DNS Data Model

Domain Name System is a hierarchical and distributed database representing the whole domain name space. The domain name space forms a tree structure and is usually depicted as an inverted tree — the root node is on the top and the leaf nodes are at the bottom. This tree is called the *zone tree* and each node in the tree is uniquely identified by a *domain name* which can contain data in a form of *resource records*. Example zone tree is shown in Figure 2.1.

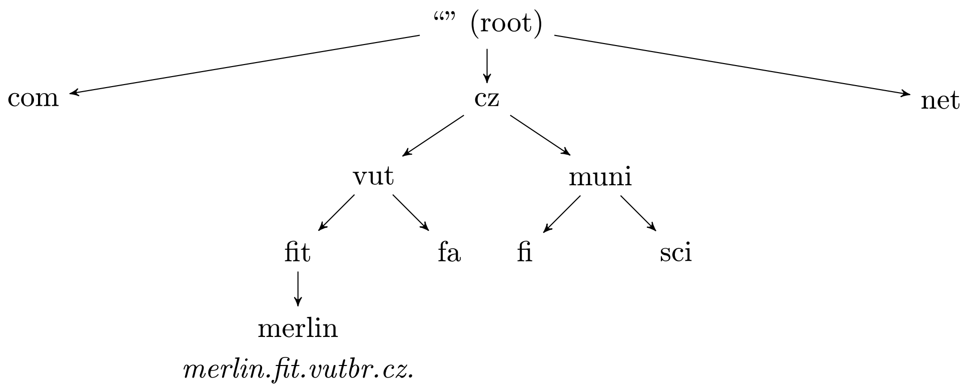

Figure 2.1: Example zone tree. Each node in the zone tree is identified by a label, a subtree within the node represents a domain. Complete domain names are built from leaves to the root node as shown on the leaf node *merlin*.

2.1.1 Domain Names

A domain name is a network resource identifier, which was designed to be independent on the network protocol family [45, 46]. Domain names are not required to include any network identifiers, addresses, routes, or similar information.

The name consists of labels separated by a dot character (ASCII code `0x2e`). Each label is 1 to 63 bytes in length, except for the root label, whose size is zero. The label corresponds with a node in the zone tree. It can contain only letters (ASCII code ranges `0x41`–`0x5a` and `0x61`–`0x7a`), digits (ASCII code range `0x30`–`0x39`), and hyphen character (ASCII code `0x2d`). The label must also begin and terminate with a letter or digit only. Originally, the labels were restricted to start with a letter, but this restriction was relaxed later [8]. Internationalized domain names (IDN) are encoded into base character set using Punycode algorithm [13]. However in reality, the rules for domain names are not obeyed strictly (e.g., the SRV records often involve labels starting with an underscore).

Complete domain names are built from the leaf nodes up to the root node. The root node is represented by an empty label. Therefore a complete domain name is terminated by a dot, however the last dot is often omitted in end-user interactions. To simplify the implementation, the total length of the domain name including the label terminators must not exceed 255 bytes.

A complete domain name uniquely identifies a network resource. The comparison of labels is done in a case-insensitive manner [17], while the DNS tries to preserve the case whenever possible [46].

Example of a domain name is `merlin.fit.vutbr.cz.`, which is composed of five labels: `merlin`, `fit`, `vutbr`, `cz`, and empty label representing the root node.

2.1.2 Domains and Subdomains

The dots in the domain names mark hierarchy levels and can be also viewed as organizational levels. These levels are called *domains*. A domain includes those parts of a zone tree, which are at the level or below the level of the domain name specifying the domain. Domains within another domain are called *subdomains*.

For example, domain name `fit.vutbr.cz` is a subdomain of `vutbr.cz`, `cz`, and root domains (as can be seen in Figure 2.1).

2.1.3 Resource Records

Each node in a zone tree includes a set of resource information. The set can be empty. Each member of the set is called resource record (RR) and has the following attributes:

- **Owner** — complete domain name of the node, where the RR is stored;
- **Type** — identifier of the type of the data held by the RR;
- **Class** — identifier of the protocol family;
- **Time to live (TTL)** — time interval the record can be kept in cache; and
- **Resource data** — variable data determined by the RR type.

Basic Resource Records

The DNS defines a lot of RR types. Some of the types from the original design were obsoleted and some new were added later. To explain the DNS data model, the following RR types have to be pointed out:

- **SOA** (Start Of Authority)

  The record specifies information about the zone authority. It contains a domain name of a primary name server, an e-mail contact of a domain administrator, a serial number of the zone content, and timing data related to zone refreshing and caching.

- **NS** (Name Server)

  The record specifies a domain name of an authoritative name server for the zone.

- **A** (IPv4 Address)

  The record specifies an IPv4 address of a host.

- **AAAA** (IPv6 Address)

  The record has the same meaning as the A record, but it is used for IPv6 addresses.

- **CNAME** (Canonical Name)

  The record specifies an alias of a domain name to another. When the DNS resolver is processing a query and reaches this record, the query is restarted and continues from the domain name pointed to by the CNAME RR data. The answer includes all resolved records, including aliases.

- **DNAME** (Delegation Name)

  The record has the same meaning as the CNAME record, but also creates aliasing for all subdomains.

- **PTR** (Pointer)

  The record has the same meaning as the CNAME record, but the query is not restarted when the processing reaches this record. PTR records are usually used for reverse address resolution (retrieving a domain name from an IP address).

Wildcard Resource Records

The DNS also defines a wildcard RR. The leftmost label of it's domain name is represented by an asterisk character (ASCII code `0x2a`). The wildcard domain name matches a domain name, if the wildcard label is removed and the remaining labels match with the suffix of the domain name being compared.

If the name server is processing a query and there is no matching record in the zone, wildcard RRs are matched. If there is a match, the name server synthesizes a new RR with the domain name from the query and the resource data copied from the wildcard RR. The synthesized record is returned in a response. The wildcard matching is limited to zone boundaries and a subdomain delegation cancels the effect of the wildcard.

The above mentioned rules imply, that the wildcard RR can match multiple labels and a better matching RR has always priority over the wildcard one.

2.1.4 Zones

The DNS is organized into units called *zones* [46]. The concept of zones allows to delegate parts of the domain name space to different authorities. Zones are also essential to achieve a true distribution of information within the DNS system. Each zone contains authoritative information for an individual domain. The highest node in the zone tree inside the zone is called the *zone origin* or the *zone apex* [23].

There is an authoritative name server for every zone. And every zone contains exactly one SOA RR in the apex node. A new child zone can be created only by a delegation, which means putting another name server in charge of a subdomain within the existing parent zone. This is done by creating NS records in the parent zone, which hold domain names of the name servers authoritative for the child zone. The place in the parent zone, where the delegation happens, is called the *zone cut* or the *delegation point* [3].

If the domain name of the authoritative name server for the child zone is located inside that zone itself, the parent zone has to include a *glue record*, which is usually an A and an AAAA record with the address of the server. The glue records are not authoritative.

Master Files

The content of the zone is specified in a textual representation in a *master file* [46]. The format of the master file is quite simple and will be explained on the following example for zone `example.com`:

```
$TTL    1h
$ORIGIN example.com.

@      SOA   ns.example.com. admin.example.com. 1234 1200 180 2419200 3600
       NS    ns.example.com.
ns     A     192.168.0.1

sub    NS    ns.sub.example.com.
ns.sub A     192.168.1.1

@      A     192.168.0.2
www    CNAME @
```

```
13 host   A     192.168.0.3
14        AAAA  fe80::3
15
16 *.test A     192.168.0.4
17 z.test A     192.168.0.5
```

The first line sets the default TTL of the RRs in the zone. The second line sets the origin domain name, which is appended to relative domain names appearing in the master file. A relative domain name is a name, which is not terminated by a dot. The symbol `@` (ASCII code `0x40`) at any occurrence stands for the current origin.

The SOA record on line 4 states that the primary name server for the zone has a domain name `ns.example.com`, the administrator e-mail address is `admin@example.com`, the zone serial is 1234, slave name servers should refresh zone content every 20 minutes (1200 seconds) and retry every 3 minutes (180 seconds) if the master server is not available. Data in the zone are valid for 28 days (2419200 seconds) if no authoritative name server can be contacted. And the last value expresses that a denying response (RR does not exist) can be cached for one hour (3600 seconds).

Line 5 specifies the only authoritative name servers for the zone, and it's IP address is specified on line 6.

Line 8 expresses the delegation of the zone `sub.example.com` to a name server with domain name `ns.sub.example.com`. Line 9 contains a glue record for the delegation, because the name server is within the delegated zone.

Line 11 states that the IPv4 address of `example.com` is `192.168.0.2` and the next line creates an alias `www.example.com` for the same host. Lines 13 and 14 specify IPv4 and IPv6 address of `host.example.com`.

Line 16 shows a wildcard RR. The server will respond to a query for an A record of `a.test.example.com`, `b.c.test.example.com`, and other similar domain names with a synthesized RR carrying IP address `192.168.0.4`. But the answer for an A record query of `z.test.example.com` will contain IP address `192.168.0.5`. The server will return a denying response to a query for `x.z.test.example.com` name.

2.2 Operation of DNS

Domain Name System uses the client–server model. In the DNS terminology, the clients are called *resolvers* and the servers are called *name servers*. The process of retrieving resource record data for a domain name is called *resolution*.

2.2.1 Types of DNS Servers

Basically, there are two types of name servers:

- **Authoritative** — provide authoritative data and non-recursive resolution; and
- **Recursive** — provide non-authoritative data and recursive resolution.

Authoritative Name Server

Authoritative name servers hold source information managed in the domain name space. Each zone has at least one authoritative DNS server, which provides the information for the whole domain associated with the zone. The authoritative name server can also delegate the authority for a subdomain to another name server.

There can be multiple authoritative servers for one zone. One of them is always a *master* server, the others are *slave* servers. All of these servers can provide authoritative answers for the maintained zone. The domain name of the master server is specified in the SOA record and every name server has an NS record. All these records are retained in the zone apex. The NS records and necessary glue records must be also present in the parent zone.

The master server stores original content of the zone. Slave servers usually use a DNS zone transfer mechanism to get an identical copy of the zone from the master.

Existence of multiple authoritative name servers for a zone increases robustness and performance of the DNS. Unavailability of one server should not directly affect operation of the other servers. Additional enhancement in these areas can be achieved by running multiple name servers as a shared unicast node [26]. This approach is used by the Internet root servers and some national domains administrators.

Non-recursive Resolution

Authoritative name servers provide non-recursive resolution only. There are three possibilities for a queried domain name affiliation with the zone:

- If the domain name belongs to the zone, the server will respond with an answer containing matching RRs.
- If the domain name belongs to a delegated zone, the server will respond with the referral information (NS and glue records).
- If the domain name lies outside the zone and any delegated zone, the server will indicate an error in the response.

Recursive Name Servers

Recursive name servers do not serve authoritative information for any zone. Their purpose is to provide recursive query resolution for other DNS clients. As their functionality overlaps with clients functionality, they are alternatively called *recursive resolvers*, *recursors*, or just *resolvers*.

Usually, these servers also implement caching functionality. RRs retrieved from authoritative servers during the processing of the query are stored for a certain amount of time, which is designated by a TTL value of the RR. The caching is sparsely implemented as a separate component and installed ahead of particular authoritative or recursive server.

Recursors are theoretically not required for the functionality of the DNS, although caching can relieve the authoritative servers of excessive load. They also make the use of *stub resolvers* possible. Stub resolvers are simple resolvers incapable of recursive resolution and are usually used in base system libraries or in embedded devices.

Recursive Resolution

The recursive resolution consist of one or more non-recursive resolutions. The first contacted server is the root server. If the server responds with a delegation, the query is resent to one of the name servers in the delegation. This server is the next server in the zone delegation chain. The process is repeated until an authoritative server for the domain name in question is reached. At that moment, the resolution is complete.

The recursive resolution is usually performed by recursive name servers or specialized libraries. A necessary prerequisite for the resolution is the knowledge of the root server IP addresses. These addresses are part of the initial cache, which is called *hints*. At least one root server has to be known by the resolver to bootstrap the access to the DNS.

2.2.2 DNS Root Zone

The content of the root zone of the Internet is maintained by Internet Assigned Numbers Authority[1] (IANA) and served by thirteen authorities [37]. Each authority is responsible for one root server node identified by a domain name in the range between `a.root-servers.net` and `m.root-servers.net`. The list of current addresses of the root nodes is distributed by InterNIC[2] licensed by Internet Corporation for Assigned Names and Numbers[3] (ICANN). Originally, there were only thirteen servers. At the moment of writing this thesis, the root zone is served by 542 servers spread around the world[4].

Top Level Domains

The root domain consists of the top level domains (TLDs), which are usually delegated to other subjects [47]. The initial TLDs are:

- world-wide generic domains — `com`, `edu`, `net`, `org`, and `int`;
- United States generic domains — `gov` and `mil`; and
- national domains — mostly two letter country codes from ISO-3166.

ICANN later approved more generic TLDs, which are currently present in the root zone (e.g., `aero`, `info`, `name`, `jobs`, etc.). Usage of some of these domains can be limited by regulations issued by ICANN. It is also very likely, that new TLDs will be added in the future. The approval process for new domains is currently going on. There are not any non-technical limitations upon new TLDs, except for domains `test`, `example`, `invalid`, and `localhost`, which are reserved to be used in documentation and for testing [22].

Infrastructure Domains

The root zone also includes special TLD `arpa`, which was intended to be used only temporarily to ease the translation from ARPANET hostnames [48, 49]. In the year 2000, the `arpa` domain was redesigned [38]. The original name derived from ARPANET is now an abbreviation for *Address and Routing Parameter Area Domain.*

[1] http://iana.org

[2] http://www.internic.net/zones/named.cache

[3] http://www.icann.org/

[4] http://www.root-servers.org

The `arpa` domain serves as a base domain for infrastructure subdomains and provides name space for translation from protocol derived hostnames to service names [32]. The most important domains within `arpa` TLD are:

- `in-addr.arpa` — translation of IPv4 addresses to hostnames [46];
- `ip6.arpa` — translation of IPv6 addresses to hostnames [10]; and
- `e164.arpa` — translation of international telephone numbers to VoIP URIs [25].

The complete list of current TLDs and related procedures can be found on dedicated website of IANA[5].

2.3 DNS Protocol

As historically there was no definition for the size of byte, all DNS standards and memos use octets as a size unit. One octet is an information consisting of eight bits. In this thesis, a byte is used when referring to size and has the same interpretation as an octet.

The term *wire format* refers to the arrangement of the data when sending them through the network. Conventionally, network byte order is used.

2.3.1 DNS Message Format

All communications within the DNS are carried in a single format called a *message*. Basic structure of the message is shown in Figure A.1.

The message has a fixed-size *Header* section, a variable-sized *Question* section, and then three variable-sized sections *Answer*, *Authority*, and *Additional*. The entries within the last three sections carry RRs and always have the same format. The size of the RRs can vary as well. The count of the entries within the variable sized sections is set in the header.

Usual DNS queries are initiated by a DNS client which prepares the message by filling the header and the question section. A name server receives the message, updates the header, keeps the question section intact, and inserts RRs into the last three sections as needed.

The header wire format is shown in Figure A.2 and the fields are explained in A.1.1, including operation and response codes. Additional response codes were defined later for the purpose of dynamic updates [56], secret key transaction authentication [57], and extension mechanisms [14].

Question Section

The Question section entry wire format is shown in Figure A.3. The entry specifies a RR to be retrieved from the server. The meaning of the fields in the question entry is explained in A.1.2.

The QTYPE field can contain a specific RR type or a special value 255 to request all RRs matching the QNAME and QCLASS.

Almost the only used class being used for RR is the Internet class (IN, value 1). Other valid classes include the Chaos class (CH, value 3), which is used by some name servers to report information about the server status [58]. There are also some other classes, but these are insignificant for the purpose of this thesis.

[5] https://www.iana.org/domains

Resource Record

The RR wire format is shown in Figure A.4. The format of the first three fields is the same as the format of the question section. The other fields are new. Their meaning is clarified in A.1.3.

Domain Names

The limitations on domain names were already described in 2.1.1. They are present in the message in the question section as QNAME field, RR sections as NAME field, and may also appear in RDATA field of some RRs.

Domain names are stored as a sequence of labels. Each label starts with one byte specifying the length of the encoded label. The domain name must be always terminated with an empty root label, which is in fact a zero byte.

As the size of a DNS message is limited, the DNS has a compression mechanism to eliminate a repetition of domain names in the message. Domain names can therefore be encoded in a compressed form. This form can be always used in QNAME and NAME fields. Compression in RDATA is usually allowed, but it depends on the individual RR type.

The compressed domain names are terminated by a *compression pointer* instead of a root label. The compression pointer consists of two bytes. The first two bits are ones to indicate the pointer, the rest of the value specifies the offset into the DNS message. The processing of the domain name continues at that offset. The sequence of labels before the compression pointer can be empty and multiple compression pointers can be chained when encoding one domain name.

2.3.2 Transporting DNS Messages

The DNS assumes that messages will be transmitted as datagrams over connectionless UDP or in a reliable stream over TCP. The service is provided on port 53 both for UDP and TCP.

UDP Transport

Datagram communication is preferred for queries due to lower overhead and better performance. The standard sets maximal size of the message to be carried over UDP to 512 bytes (on application layer [9]). If the message is larger than this limit, the name server sets TC (truncated) flag in the message header and provides only a partial answer.

TCP Transport

TCP is used for messages not fitting into the UDP size limit or when a reliable transfer is required (e.g., for zone transfers). Transmitted DNS messages are prefixed with a two byte length field with the length of the message, which will follow. This creates a size limitation of 64 kilobytes for a message on TCP. This is believed to be sufficient for regular queries and there is no mechanism defined to handle larger answers.

As for zone transfers, the data are usually larger than TCP message limit and therefore the transfer is split into multiple messages. The first and the last messages are recognized due to a presence of the same SOA RR [43].

2.3.3 Extension Mechanisms for DNS

The original DNS protocol includes a number of fixed fields whose range does not allow the protocol to grow. It also puts excessive limitations on DNS message size for datagram transport and does not allow advertising of capabilities between communicating sites. To solve these problems, the Extension Mechanisms for DNS (EDNS) was introduced [14]. It provides backward-compatible mechanism for extending the protocol. The EDNS is a necessary prerequisite for DNS Security Extensions (DNSSEC).

The EDNS defines two communicating sites — *requestor* and *responder*. The requestor is the side that sends a request, the responder is the name server or other DNS component that responds to questions. The EDNS is a hop-by-hop extension, which means that the EDNS content in the message is settled for each pair of hosts involved in the resolution process.

To provide the backward compatibility, the EDNS introduces pseudo RR type OPT, which may be added into the additional data section of the request. If a compliant resolver receives such request, it must include the OPT record in the response as well. Since the OPT record does not carry any DNS data and only contains control information for the processed transaction, it must not be cached, forwarded, or stored in a master file.

The wire format of OPT RR matches the wire format of other RRs and therefore can be parsed by EDNS non-compliant host. Yet EDNS specifies a different meaning of some fields to save a space in the message. The new meaning of the fields is described in A.1.4.

The following fields are essential for the DNSSEC operation:

- **EDNS version**

  The requestor should always set version to the lowest implemented level capable of expressing a transaction to maximize compatibility with the responder. On the other hand, responder should set the value to the highest available implemented version, but not higher than the value received in the query. This allows the requestor to learn a maximal supported version of EDNS if the query cannot be resolved because unsupported EDNS version.

- **Maximal Payload Size**

  The maximal payload size in the OPT record specifies the maximal size of a DNS message in bytes, which can be reassembled and delivered by the network stack of the communicating part, which sent the message. Values lower than 512 are treated as equal to 512 (a maximal message size without EDNS).

- **EDNS flags**

  EDNS(0) specifies only one flag called DNSSEC OK (DO) [11]. Setting this bit in the query indicates, that the requestor is able to handle DNSSEC security RRs. Otherwise, security RRs are not included in the response unless explicitly queried for. Responder copies the value of the bit into the response.

Chapter 3

Domain Name System Security Extensions

Domain Name System Security Extensions (DNSSEC) is a mechanism for assuring integrity of DNS data and authentication of its origin. The chapter summarizes security threats DNSSEC can prevent, an application of public-key cryptography in DNSSEC, new resource record types introduced by DNSSEC, and a procedure of securing of a DNS zone.

In 1990, Steven Bellovin wrote an article about abusing the DNS system for system break-ins [7]. The publication of the article was withheld for over four years by the author, because it described serious vulnerabilities, for which there there was no feasible fix. However the secrecy appeared to be pointless, as the article leaked anyway and there were reports that the same technique was used by hackers earlier. Former publication might have helped to find the solution sooner. The article describes two problems:

1. **Host name based authentication**

   At the time, the software used to employ the hostname based authentication. When the client initiated a connection to a server, the server took the client's IP address and translated the IP address to a hostname. If the obtained hostname was on the list of allowed hosts, the client was authenticated.

   This worked fine with static hostname mappings. But the DNS uses two separate trees — the first for name-to-address mapping (starting in the root domain) and the second for address-to-name mapping (starting in `in-addr.arpa` domain). The consistency of these trees is not a matter of the DNS. As the reverse mappings are in control of the IP address owner, they could be modified to return a false domain name and thus pass through the authentication.

   The fix for this vulnerability was simple. Applications performing a hostname based authentication were modified to check both reverse PTR and forward A RRs. If the hostnames in the records did not match, the authentication failed. However the hostname based authentication is fundamentally flawed and authentication based on cryptography should be used instead. At the time, Kerberos authentication mechanism was already available.

2. **DNS Additional Section Handling**

   The described problem resides in the handling of DNS messages by caching name servers. The RRs in the additional section of DNS messages were usually put into the cache without checking. But the content of the additional section must be carefully investigated before acceptance.

An attacker can ask a caching name server (or can persuade another host to do so) for an arbitrary record within a subdomain of a domain controlled by the attacker, say `host.sub.attacker.test`. During the recursive resolution, the authoritative name server for domain `attacker.test` will return a response with a list of name servers authoritative for the subdomain. These records might be accompanied with glue records. The attacker can include a victim's hostname in the list of name servers with fraud A glue record. Accepting this glue record results in a DNS cache poisoning.

Hijack of InterNIC Domain

In the year 1997, Eugene Kashpureff managed to hijack InterNIC domain and redirect it to his own AlterNIC domain for a few days [44]. He abused a vulnerability in BIND name server, which allowed poisoning of the cache by placing any RRs into the additional section of DNS messages. As a result, concept of *bailiwicks* was introduced [59]. It specifies domain names for which RR in the additional section can be accepted.

The other way of causing a cache poisoning is to forge an answer from a name server. The validity of a DNS response is verified by checking the random transaction ID in the header of the message and sometimes by checking the question section. The attacker must guess all these information and send the bogus packet with a proper timing. If the attacker is not successful, the correct answer will be cached and there is no chance to attack the same hostname until the TTL of the cached response expires.

Bypassing Bailiwicks

In the year 2008, Dan Kaminsky explained how to bypass the bailiwicks and the data stored in the cache, thus allowing the attacker to try to poison the cache at any time [36]. The possibility to perform the attack much faster makes cache poisoning very dangerous.

Let's say the attacker wants to insert a wrong A RR for domain `www.victim.test` into the resolver's cache. The attacker asks the resolver for non-existing domain within the same domain (e.g., `nx-0.victim.test`) and immediately sends a fake response, which contains a NS record in authority section pointing to hostname `www.victim.test` and a glue record in the additional section with a fake address.

If the attacker is successful in guessing the transaction ID and the fake response is delivered before the correct one, the attack is successful. As all the domain names in the packet belong to the same bailiwick, the glue record will be added into the cache (possibly replacing existing entry). If the packet is not accepted by the server, the attacker can immediately repeat the attack by asking for another non-existing domain name (e.g., `nx-1.victim.test` by increasing the counter).

As a defence against Kaminsky's attack, the resolvers implemented randomization of source ports. The port number is used in addition to transaction ID to check the validity of the response. The attacker's odds to guess a transaction ID are $1 : 2^{16}$, with added port number the odds drop to $1 : 2^{32}$.

IP Fragmentation Attack

In the year 2012, Amir Herzberg and Haya Shulman drew attention to a possible cache poisoning attack based on the IP fragmentation [29]. The attack has relatively low technical complexity, but requires on a lot of preconditions to be satisfied [30].

Basically, the attacker needs to persuade an authoritative name server to send a fragmented response to a query initiated by a recursive name server, whose cache the attacker intends to tamper. The fragmentation can be enforced by sending a spoofed ICMP packet to the authoritative server or by using a specially constructed RR in a valid zone controlled by the attacker. Ideally, a message will be split into two fragments and the attacker aims to forge the second one.

As the UDP and the DNS header will be always in the first IP fragment, the attacker need not to guess the DNS transaction ID nor the query source port. The problem reduces to guessing of a 16-bit IP datagram identification. In addition, the UDP packet checksum must be left intact, which is relatively simple due to the used checksum algorithm.

3.1 DNSSEC Threat Analysis

The first version of DNSSEC has been designed without a specific list of threats against which DNSSEC should protect. Therefore it was not possible to verify that it meets it's design goals. The detailed analysis was written in 2000 [5]. The informational memo states some basic assumptions about DNSSEC:

- Data in the DNS are public. DNSSEC offers no protection against data disclosure.
- Client authentication is out of scope of DNSSEC.
- The solution is backward compatible and can coexist with secure unaware DNS components.
- DNSSEC provides data integrity and data origin authentication.
- Digital signatures are used to support the desired services.

DNSSEC takes following threats into account:

- **Packet Interception**

  Packet interception includes monkey-in-the-middle attacks, eavesdropping combined with spoofed responses. Unencrypted UDP packets make this kind of attacks quite easy. DNSSEC does not provide any encryption, because it would bring high processing cost per message and also for establishing and maintaining a bilateral trust between all parties. In addition, this would provide only hop-to-hop security, not end-to-end security. DNSSEC offers end-to-end data integrity check, which is more useful for DNS resolutions.

- **ID Guessing and Query Prediction**

  This kind of threat is very similar to Packet Interception, except that the attacker does not have to be on a transit or a shared network. On the other hand, the attack works only if the attacker is able to guess the transaction ID of the message and the source port used by the client. Theoretical maximal amount of combinations for these values is 2^{32}, although practically the search space is smaller because transaction IDs and port numbers can be predicted from previous traffic or from known state of the client. Sometimes the attacker can even influence these allocations.

- **Name Chaining**

  Name Chaining attacks are subset of cache poisoning attacks. This kind of threat involves RRs which have domain names in RDATA field. Modification of these RRs lets the attacker to feed the victim's cache with bad data or subvert subsequent decisions based on these names. Worst examples in this class of RRs are CNAME, DNAME, and NS RRs, as they can be used to redirect the victim's query. RRs like MX and SRV can be modified to trigger further DNS lookups or redirect new connections to services. As a corner case, address records like A and AAAA can be used in these attacks — these records do not have domain names in RDATA, but can be used in reverse address resolution.

- **Betryal by Trusted Server**

  This threat is very similar to Packet Interception, except that the client sends the packets to the attacker voluntarily. The manipulation with DNS messages can be caused by wrongly configured resolvers, which may use other untrusted name servers to resolve the queries on their behalf. Or the messages can be modified intentionally by the internet service provider, regardless the purpose is legitimate or not.

- **Denial of Service**

  DNSSEC does not offer protection against Denial of Service attacks. In fact, checking of signatures increases processing cost of each message and in some cases increases the number of messages needed to answer the query. Therefore DNSSEC makes the problem worse. In addition, signed responses are much longer than originating queries, which puts name servers as risk of being used as amplifiers for this kind of attacks.

- **Authenticated Denial of Domain Names**

  Absence of RR which causes an action other than immediate failure constitutes a real threat, although severity of these threats is arguable. This is a case for MX, SRV, and probably other type of records. DNSSEC includes mechanism, which can prove which domain names exist in the zone, and which RR types exist at those names.

- **Wildcards**

  The processing of wildcard RRs is relatively complicated and therefore creates some risk itself. The applicability of wildcard synthesis rules must be verified. This can be done by proving the existence of the wildcard record itself and also by proving non-existence of RR for the exact name in a query. Therefore this mechanism is dependent on Authenticated Denial of Domain Names.

DNSSEC can be supported by transaction security, which is not a part of DNSSEC, but was designed in parallel to it. Transaction security provides authentication between two directly communicating components of the DNS. That can be used to authenticate dynamic updates (DDNS), zone transfers, and to secure communication between a stub resolver and a trusted validating resolver. Multiple mechanisms providing transaction security are available:

- **TSIG** — uses one way hashing and shared secret [57].

- **SIG(0)** — depends on public-key cryptography [16].

- **GSS-TSIG** — utilizes Generic Security Service API [40] (e.g., Kerberos).

3.2 Public-key Cryptography

DNSSEC relays on *public-key cryptography* (also called *asymmetric cryptography*). It uses a concept of two separate signing keys. The first key is called *public* and the second is called *private*. Both keys are mathematically bound and can be used only for complementary operations — if a public key is used to encrypt a message, only a corresponding private key can be used to decrypt the cypher text and vice versa.

The name of the key matches with the secrecy, in which the key must be kept. The private key should be known only by the owner of the key because some encryption algorithms allow to derive the public key from the private one.

Basically, public-key cryptography can be used for:

- **Secrecy**

  Public-key cryptography can ensure confidentiality between to communicating parties. Each party must have a pair of a private and a public key. Private keys are known only by the key owners. Public keys are known by everyone (including any third party).

  The messages to be transfered over an insecure channel are usually encrypted twice. The first encryption is done using the private key of the sender, the second encryption is performed with a public key of the receiver. The encrypted message can then be decrypted only by the receiver, again in two decryption cycles – first by the private key of the receiver, then by the public key of the sender.

  Technically, only the encryption using a public key of the receiver is sufficient to ensure the secrecy of the transfered message. The purpose of the first encryption cycle is to prove the identity of the sender. This mechanism is safe only if both parties are sure about the ownership of the public keys.

- **Digital signatures**

  Public-key cryptography can also ensure authenticity, integrity, and non-repudiation. In this case, the mechanism is usually supported by a one way hashing function.

  A hash of the message to be signed is computed. This hash is then encrypted with private key of the signer and the resulting cypher text is the digital signature. The digital signature is usually attached to the message. Anyone who knows the public key of the sender can decrypt the digital signature, recompute the hash of the original message, and compare the results. If both hashes are the same, the message was not modified and must have been signed by the sender, because we assume that the sender is the only owner of the private key.

- **Key agreement**

  Certain algorithms can be also used for key agreement, where two parties can establish a shared secret key over an insecure communication channel. The key can be then used for encryption using symmetric cryptography.

  First, both parties exchange their public keys. Then each party uses it's private key and the public key of the other party to compute the new key value. The resulting key value is the same on both sides.

In order to use public-key cryptography effectively, a *Public Key Infrastructure* (PKI) is required. It is a system of certificate authorities, that perform certificate management, key management, and token management functions for a community of users in an applications of asymmetric cryptography [52]. It includes hardware, software, people, policies, and procedures.

3.3 DNSSEC Resource Records

DNSSEC builds a subset of PKI on top of existing DNS infrastructure. PKI works with certificates, which are objects that truly identify an owner of a public key. Validation of certificates is performed by certification authorities (CA), which prove the validity of the certificate by digitally signing the content of the certificate using the CA's key. The CAs posses their own certificates as well, and these certificates are usually signed by other CAs. The validation of a certificate is performed by following the chain of signatures in the certificate, until a valid signature of a trusted CA is found.

DNSSEC introduces several new RR types, which allow to store public keys in the zone and digitally sign the content of the zone. The presence of the public key in the zone associates the key and the zone. In a zone delegation, parent public key can be used to sign the key for the child zone using the same mechanism. Some public keys have to be set in the resolver configuration as *trusted anchors* and are used as a starting point when doing the validation. The trusted anchor and public keys successfully validated at delegation points create a *trust chain*.

3.3.1 Initial Version of DNSSEC

The design and deployment of DNSSEC was quite slow. The initial draft was published in 1997 [21] and the final version of DNSSEC was published in 1999 [18]. At that time, BIND 9 was the first and only DNSSEC capable name server implementation. However the deployment of DNSSEC was stalling.

DNSSEC added three new RR types: KEY for storing public keys, SIG for storing signatures, and NXT as an auxiliary RR required for authenticated denial of domain names.

Each DNS zone is usually managed by a different authority, therefore every secured zone was expected to contain a KEY RR, which holds a public key for validation of signatures within that zone. To achieve a validation across delegation points, the KEY RR in a child zone must be signed by a private key of it's parent zone. This signature must be present in the child zone, otherwise the signature would not be considered authoritative. The parent zone can optionally contain the key and signature for the child zone as well. However, if a secure zone contained insecure delegations, the zone had to include a special null key and it's signature for each of these insecure delegations.

The experiments showed that this mechanism carried a lot of problems, mostly with key handling. When a zone administrator needed to change a signing key, the new key had to be submitted to a parent zone administrator, the new signature was submitted back, and finally inserted into the zone. In addition, the change of a signing key required to re-sign all keys for every delegation in the zone and to publish the new signatures in the child zones, which was found almost impossible, especially for TLDs. If the child zone got out-of-sync from it's parent zone, resolvers could recognize the RRs in the child zone as bogus.

3.3.2 Second Version of DNSSEC

Attempts to solve individual problems of DNSSEC were not very successful and finally, a new version of DNSSEC was created, initially called DNSSEC-bis. The final version was published in 2005 in three separate documents [2–4] and made all the old DNSSEC memos obsolete.

The new DNSSEC is incompatible with the old one, but is very similar. It simplifies the RRs for keys and solves the problems with key handling at delegation points by introducing another RR type DS, which certifies the signing key for a child zone but is kept in the parent zone.

Resource Record Set

The granularity of signatures in DNSSEC is limited to *resource record sets*. A RR set is a group of RRs which belong to the same domain name and have the same type, class, and TTL value. Originally, the TTL value could be different for each record within the set, but it was causing partial replies from caching name servers and these replies did not indicate truncation. Because of this, different TTLs in a RR set were deprecated [23].

Canonical Representation

The signature validation requires data to be in the exactly same format and order as when they were signed. However, ordering of RRs in the DNS is irrelevant, domain names are case insensitive, and also domain name compression applies during the transfer. For the purpose of signing, DNSSEC defines the canonical representation of domain names, RRs, and their sets.

The conversion to the canonical representation consists of the following steps:

1. Domain names are ordered by individual labels, starting with the most significant (rightmost) ones. First, all uppercase ASCII letters are converted to lowercase and then each label is treated as unsigned left-justified byte string. The absence of a byte sorts before a zero value byte. If two labels match exactly, the sorting continues with next less significant labels.

2. All domain names in the RR (owner and RDATA fields) must be absolute, without compression, and all uppercase ASCII letters in them are converted to lowercase. If the domain name is a wildcard name, it has to be in its original form with no substitution. The TTL field is also set to it's original value.

3. The RRs within a set are sorted by treating RDATA field of the canonical form of each RR as a left-justified sequence of unsigned bytes, while absence of a byte sorts before a zero byte.

DNSSEC Resource Records

The DNSSEC documents add following RR types into the DNS:

- **DNSKEY** RR holds a public key. It is not intended to use this type of record for storing arbitrary public keys, but only keys that directly relate to the DNS infrastructure. The public key can be used by a validating resolver to authenticate RR sets.

The format of DNSKEY RDATA is shown in Figure A.7, the meaning of the fields is explained in A.2.1.

- **RRSIG** RR contains a digital signature of a RR set. It also specifies a validity interval of the signature and information necessary to identify the key used for signing. The name and class of the signed RR set is determined by the name and class of the RRSIG RR itself, type and TTL value are included in RDATA. The format of RDATA is shown in Figure A.8, the meaning of the fields is explained in A.2.2.

- **NSEC** RR is required for authenticated denial of domain names. The record is created for all existing domain names in the zone, which has authoritative data or a delegation point. It lists two domain names and proves, that no other domain name exists between these two names in canonical ordering of the zone. The format of the record is shown in Figure A.9, the meaning of the fields is explained in A.2.3.

- **DS** (Delegation Signer) RR refers to a DNSKEY and is used to authenticate that DNSKEY during delegation. The pair of DS and DNSKEY records have the same owner name, but are stored in different locations. The DS record is stored in the parent zone, the DNSKEY is stored in the child zone. The records are considered authoritative at these locations. The format of the DS record is shown in Figure A.10, the meaning of the fields is explained in A.2.4.

NSEC3 Resource Records

The NSEC records meet the requirements for authenticated denial of existence, however it make the zone content enumerable, which was not possible prior DNSSEC. One NSEC record lists two names that follow each other in the canonical ordering of the zone. All NSEC records in the zone create a closed NSEC chain. By querying for the NSEC record of the zone apex and by following the names in Next Domain Name field, the zone content can be enumerated trivially. The enumerated zone can be used to reveal registrant data, which many registrars may have legal obligations to protect. Use of NSEC renders these policies unenforceable.

As a solution for this problem, NSEC3 has been introduced [41]. In principle, it uses cryptographic hashes instead of domain names. In addition, it eases the manipulation with NSEC chain for unsecured delegations. NSEC records must exist for each delegation in the zone and the NSEC chain has to be updated with every change in these delegations. NSEC3 introduces *opt-out*, which allows to exclude insecure delegations from the NSEC3 chain. Addition, removal, or modification of insecure delegation then have no effect on existing NSEC3 chain.

NSEC3 adds two more RR types:

- **NSEC3** RR has the same meaning as NSEC RR. The record must exist for each domain name in the zone, which contains authoritative data. If opt-out is not used, an NSEC3 RR must also exist for each domain name containing only an insecure delegation.

  For the purpose of NSEC3, the domain names (called *original*) are cryptographically hashed and used to construct new domain names (called *hashed*). The hash of the original domain name is encoded in *base32hex* [34]. By appending a zone apex to the encoded hash, a domain name to be used as an owner of the NSEC3 RR is constructed.

The NSEC chain is created by canonical ordering of original domain names, NSEC3 chain is created on the same principle but from hashed domain names.

The format of the NSEC3 RDATA is shown in Figure A.11, the meaning of the fields is explained in A.2.5.

- **NSEC3PARAM** RR has the same format as the initial portion of NSEC3 record, which can be seen in Figure A.12. It should contain the same values as used in NSEC3, but the Flags field has to be zero (opt-out flag must not be set). The record is always present in the zone apex and serves as a hint for the authoritative name server to use NSEC3. It is not used by validators nor resolvers.

3.4 Securing the Zone

The process of securing a zone using DNSSEC is called *zone signing*. Zone signing requires certain operations to be performed in the zone and may also require changes of delegation in the parent zone.

DNSKEY Resource Records

In order to sign a zone, one or more signing keys have to be generated. The DNSSEC validation protocol does not distinguish between different types of DNSKEYs, but usually following roles of keys are differentiated [39]:

- **Key Signing Key** (KSK) is used to sign DNSKEY resource recods in the zone apex only. KSK DNSKEYs usually have Secure Entry Point flag set in RDATA.
- **Zone Signing Key** (ZSK) is used to sign all other records in the zone.
- In **Single-Type Signing Scheme**, all keys are used to sign all records, regardless their type.

The reasons to separate KSK and ZSK roles are purely operational. KSKs are used as trust anchors and any change of the trust anchor is relatively complicated because interaction with the parent zone is necessary. As the change spawns across multiple zones, it is slower because of multiple levels of caching. In case the parent zone is not secured, change of the KSK requires reconfiguration of resolvers.

KSKs are also used less frequently, which makes it possible to store the private keys offline or enforce other stronger security policy. Once a DNSKEY RR set is signed, all keys in the set can be used as ZSKs. These keys are equally valid. If a ZSK is compromised or if there is another reason to change it, the rollover is much simpler because only a change within the zone is needed.

The zone should include all DNSKEY RRs used to sign the zone. In certain situations it may contain no DNSKEYs (when the zone is used as an island of security and the keys are configured on resolvers), or DNSKEYs which have not been used for signing yet (key pre-publishing for faster key rollovers).

RRSIG Resource Records

The signature has a form of RRSIG RR. Each authoritative RR in a zone must be signed, except for other RRSIG records. Each of these records must be signed using at least one DNSKEY of each algorithm in the DNSKEY RR set in the zone apex.

The validating resolver performs the validation by taking all RRSIGs of a covered RR, and tries to validate at least one signature by some matching and trusted DNSKEY. If a key, for which the signature is valid is found, the result of the validation is a success. Otherwise, the signature is considered invalid.

These definitions creates important assumptions for key and algorithm rollovers. In case of a key rollover, keys and signatures can be published independently. In case of algorithm change, the signatures must be always pre-published without a public key.

NSEC Resource Records

For authenticated denial of existence, a complete NSEC or NSEC3 chain has to exist in the zone. During NSEC to NSEC3 rollover, both chains may exist simultaneously.

The NSEC record is created for every domain name holding authoritative data and also where the NS records for delegation reside. They must not be created for domain names, which do not have any RRs. The bitmap consists only of authoritative RRs present at the name (including NSEC and RRSIG) and NS record in case of delegation point. No glue records are included.

The TTL value of NSEC RRs must be equal to minimum TTL in the zone SOA RR.

NSEC3 Resource Records

The NSEC3 record is created for every domain name, where authoritative data or secure delegation NS records exist. The insecure NS records need not be covered by NSEC3, but this actuality must be indicated by the opt-out flag. The record must be also created for empty non-terminal owner names, which lead to another NSEC3 covered domain name. An empty non-terminal can emerge for example when the zone contains RRs with `c.test` and `a.b.c.test` owners, but no RRs with `a.b.test` owner. The bitmap must contain only authoritative RRs present at the original name. As NSEC3 records are stored as hashed names, the bitmap will never contain NSEC3 record and may not include RRSIG record, if there are no other authoritative records at the original name.

Internally, NSEC3 domain names should be kept separately from other domain names. Therefore there is no limitation on creating RRs with owner names colliding with NSEC3 hashes. As a result, explicit queries for NSEC3 records are not possible.

The TTL value of NSEC3 record is set on the same basis as for NSEC records.

NSEC3PARAM Resource Records

The NSEC3PARAM RR is present only in the zone apex and indicates, that the authoritative server should use NSEC3 for authenticated denial of existence, instead of NSEC. Flags field of NSEC3PARAM must be zero, otherwise the record should be considered invalid. The existence of the record has no meaning for validation.

DS Resource Records

The DS RR is used to establish a connection in a trust chain between a parent zone and a child zone. It refers to a DNSKEY used as a secure entry point for the zone (KSK or any other key in case of Single-Type Signing Scheme). The record is generated from the DNSKEY and submitted to the parent zone administrator. The record is then inserted into the parent zone and signed by the parent zone DNSKEY.

The DS and NSEC records are the only records, which are authoritative in the zone at a delegation point.

3.5 Proving the Answer Validity

DNSSEC requires EDNS for DO flag (DNSSEC OK) and to allow transport of larger messages over UDP without truncation. DNSSEC also defines two new flags AD (Authentic Data) and CD (Checking Disabled) in the DNS message header, as shown in Figure A.2. The flags have the following meaning in DNSSEC:

- The **DO flag** is set in the query and indicates, that the requestor is able to handle DNSSEC security RRs. The responder copies the value of DO flag into the response message.

  If the DO flag is set, the response must include all RRSIG records to authenticate the answer, necessary NSEC or NSEC3 records if the response is a denial of existence, and DS records in case of a zone delegation. These records are placed in the Authority section. Optionally, DNSKEY records for the zone apex can be included in the Additional section, but their inclusion must not cause a DNS message truncation.

  In case the DO flag is not set, security RRs are not included in the response unless explicitly queried for (with the exception for NSEC3 records).

- The **AD flag** is controlled by a name server and ignored in queries. Setting of this flag indicates, that the responding name server considers the data in the answer and authority sections to be authentic.

  Security aware authoritative name server can set this flag (even without checking the signatures), when explicitly configured by a system administrator. Otherwise it should clear this flag.

  Security aware recursive name server must set this flag only if it can fully authenticate the RRs by validating the RRSIGs. In case of answering from insecure zone, the flag must be cleared. If the validation fails on secure zone, the server returns server failure RCODE in the message header.

- The **CD flag** is controlled by resolver and name server copies the bit into the response. Setting this flag disables signature validation in security aware name servers.

As the AD flag can be modified by an attacker, the flag should not be trusted. In order to validate a RR, the resolver has to build and validate a whole chain of trust. In case of a stub resolver, the forwarding resolver needs to be trusted and so the connection to it.

Chapter 4

Existing DNSSEC Implementations

This chapter analyzes existing open-source solutions for management of DNSSEC on an authoritative server side. The analysis is limited to identification of supported features, supplied tools, and different approaches for the DNSSEC signing. As the results will be used during the design of the new DNSSEC library, the analysis tries to highlight advantages and drawbacks of these solutions, especially from the perspective of a zone operator.

4.1 BIND

BIND (Berkeley Internet Name Domain)[1] is the first DNS name server implementation, which was written at the University of California as a result of DARPA grant [53]. Later, the project was taken over by Paul Vixie, founder of *Internet Systems Consortium* (ISC). ISC maintains and develops BIND until now.

At the moment, BIND versions 9 is actively being used and developed. BIND 9 was started in 2000 and fully supports DNSSEC. It provides tools for key maintenance, zone signing, and validation. It supports NSEC3, key rollover, automatic signing of dynamic zones, DNSSEC Look-aside Validation (DLV), and automatic updates of trust anchors.

The ISC also worked on BIND version 10, but this project was discontinued in April 2014 due to a lack of resources [50]. Consequently, the BIND 10 was renamed to Bundy[2] and left for the open-source community. Bundy uses brand new modular architecture and is designed more as a framework. Although it brings a lot of new features (like support for SQL backend), the project is at the time of writing this thesis not mature and a lot of features are incomplete. The DNSSEC support in Bundy is limited to serving of pre-signed zones only. In the following text, BIND 9 is meant when referred to BIND without explicitly stating the version.

BIND provides a few elementary utilities, which are necessary for DNSSEC signing:

- **dnssec-keygen** generates a new DNSSEC key pair. Two files are created as a result. E.g., `Ktest.com.+007+56463.key` and `Ktest.com.+007+56463.private`. The part `test.com` is the zone name, `007` identifies the algorithm (in this case RSA with SHA1, capable of NSEC3), and `56463` is the key tag of DNSKEY. The `.key` file contains a

[1] https://www.isc.org/downloads/bind/

[2] http://bundy-dns.de/

public key as a DNSKEY RR in the master file format, the `.private` file contains a private key in a custom BIND format and optionally DNSKEY timing metadata.

- **dnssec-dsfromkey** generates DS record set from existing keys. The record is printed in the master file format.
- **dnssec-settime** changes timing metadata of an existing key. BIND specifies a publication date (when the DNSKEY is added into the zone), activation date (when the key is started to be used for signing), inactivation date (when the key is stopped being used for signing), deletion date (when the key is removed from the zone), and optionally revocation date (when the self-signed key is added into zone with a revocation flag set).
- **dnssec-revoke** is a one-purpose utility for setting a revocation flag of an existing DNSKEY. From the view of BIND, it means creating a new key because revocation flag causes a change of its key tag.
- **dnssec-signzone** generates NSEC or NSEC3 chain, computes signatures, and produces a newly-signed master file. The keys to be used for signing must be specified explicitly or a directory with set of keys can be provided instead. The utility will sign the zone with all keys, whose DNSKEYs are present in the master input file, unless smart signing is used. The smart signing mode utilizes timing metadata of the keys — it updates DNSKEY records in the zone and performs the signing with active private keys only.

The process of key generating is manual and setting the timing metadata can be tricky. Therefore BIND includes another set of tools, which can help to minimize potential problems:

- **dnssec-verify** verifies validity of signatures and completeness of a NSEC chain for a given zone master file.
- **dnssec-checkds** retrieves DS records for a given zone from the DNS and checks whether the records successfully verify given signing keys.
- **dnssec-coverage** analyzes given keys for one zone. The tool prints all upcoming key events based on the timing metadata and informs about problems, which can cause breaking of the chain of trust.

BIND also supports signing keys stored in hardware security modules (HSM) for the purpose of private key protection or signing acceleration. The access to the keys is provided via PKCS #11, which is a cryptographic token software interface standard [51]. BIND includes following tools to cooperate with HSMs:

- **pkcs11-keygen** generates new key pair in the PKCS #11 device.
- **pkcs11-list** lists all objects in the PKCS #11 device and their identifiers (label and ID).
- **pkcs11-destroy** erases all objects in the PKCS #11 device.

- **dnssec-keyfromlabel** creates BIND compatible key files from a key stored in PKCS #11 device. The source key is identified by the label. The created public key has the same format as a public key generated by *dnssec-keygen*. But private keys on the HSM are sensitive and usually cannot be exported, the new private key file contains just a reference to the key on the HSM device and timing metadata.

With the help of these tools, BIND can operate in following DNSSEC modes:

- **Serving of pre-signed zone**

  This is the simplest case. BIND is given a pre-signed master file produced by the *dnssec-signzone* utility. Configuration for the zone contains no special setting:

  ```
  zone "example.com" IN {
     type master;
     file "zones/example.com.signed";
  };
  ```

  In this case, the zone must be static. The zone is under full control of the zone administrator, who is required to re-sign and reload the zone regularly to keep the signatures valid. This is also the only mode supported by BIND 10 and some other simple DNSSEC-aware name server implementations.

- **Automatic signing using Dynamic DNS update method**

  In this configuration, BIND must be aware of the location of the signing keys, as the signing keys can be exchanged at runtime using DDNS. This will reflect in the zone configuration:

  ```
  zone "example.com" IN {
     type master;
     file "dynamic/example.com";
     update-policy local;           # enable DDNS
     key-directory "keys";          # signing keys location
  };
  ```

  BIND provides *nsupdate* utility for sending DDNS updates. For example, to add another ZSK, following commands can be issued in *nsupdate* prompt:

  ```
  > server 127.0.0.1 53
  > add example.com. 3600 IN DNSKEY 256 3 7 AwEAAZwcROGiozZymHD3AJV5k...
  > send
  ```

  The server will add the DNSKEY and if the private key is available in the key directory, new signatures will be added. This method can be also used to configure NSEC and NSEC3. To change NSEC3 hashing parameters, following update can be sent:

  ```
  > delete example.com. IN NSEC3PARAM
  > add example.com. 0 NSEC3PARAM 1 1 10 C01DCAFE
  ```

In this case, all existing NSEC3PARAM records are removed and a new one is created. If no NSEC3PARAM records are in the zone, BIND will use NSEC for authenticated denial. Also note, that setting the opt-out flag in NSEC3PARAM is not valid. In fact, BIND will add a correct NSEC3PARAM without the flag into the zone, but the flag will be set in NSEC3 records and insecure delegations will not be covered.

This configuration provides the same functionality as the *dnssec-signzone* non-smart signing, which is virtually performed after each DDNS change. To request an immediate re-signing of the zone, use the BIND server control utility:

```
$ rndc sign example.com
```

Also, BIND stores all dynamic changes including related DNSSEC records in it's own binary format for better performance. To write down the changes to a master file, use:

```
$ rndc flush
```

- **Fully automatic zone signing**

  An improved version of the previous solution is the fully automatic zone signing. It can be enabled by adding `auto-dnssec` option into the zone configuration:

  ```
  zone "example.com" IN {
     type master;
     file "dynamic/example.com";
     update-policy local;
     key-directory "keys";
     auto-dnssec maintain;         # or: allow
  };
  ```

  The possible values of `auto-dnssec` are `allow` and `maintain`:

  - `allow` value: In this case, the records in a zone are being re-signed in a smart way when a DDNS update is received and when `rndc sign` command is invoked. Thus in addition to previous DDNS solution, DNSKEY records are maintained and signatures generated automatically according to timing metadata.
  - `maintain` value: Behaves the same as `allow`, but does not wait for `rndc resign`. The zone is signed automatically when loaded for the first time and whenever necessary — based on the timing data. The key directory is rescanned for new keys metadata once an hour (can be customized), or when `rndc loadkeys` is invoked.

The maintenance of a DNSSEC zone using BIND in the most advanced mode is quite easy, but a lot of manual interventions are still necessary. The keys have to be generated regularly and their timing metadata have to be set correctly. And any manual action can lead to configuration errors rendering the zone invalid. However, the tools for detecting these situations are included.

4.2 NSD

NSD (Name Server Daemon)[3] is a simple, high performance, and authoritative-only name server by NLnet Labs. Under the hood, it uses the ldns[4] library from the same authors.

The server supports DNSSEC pre-signed zones only. Therefore it cannot be used for serving secured dynamic zones. The zone administrator is also responsible for generating of the keys and keeping the signatures up to date. The utilities for DNSSEC are not provided by NSD, but ldns library. Some of the utilities are heavily inspired by utilities from BIND, but may not provide perfectly equivalent functionality:

- **ldns-keygen**, **ldns-signzone**, **ldns-revoke**, and **ldns-key2ds** are tools created as a replacement for *dnssec-keygen*, *dnssec-signzone*, *dnssec-revoke*, and *dnssec-dsfromkey*. The generated keys use the same format as keys used by BIND, which allows the usage of BIND generated keys with NSD and vice versa. However timing metadata and PKCS #11 keys are not supported by any ldns-based tool.

- **ldns-test-edns** checks if a target server supports DNSSEC by sending it a query with DNSSEC OK flag set in EDNS.

- **ldns-walk** retrieves all domain names in a zone by performing NSEC enumeration. Of course, this is possible only on DNSSEC signed zones which use NSEC.

- **drill** is a full featured DNS lookup utility inspired by *dig* from BIND tools. Besides other things, *drill* allows signatures chasing (printing the whole DNSSEC trust chain from the root domain to the last RRSIG), which is very useful when debugging problems in DNSSEC validation.

There are also other utilities included in ldns, but are not related to DNSSEC or provide just a subset of functionality provided by other tools, especially by *drill*. Many of these utilities were initially created to demonstrate functionality of ldns library and were not intended to be used in a real life.

NSD was designed as a lightweight and fast server and combined with the tightly bound ldns utilities creates a self-sufficient solution for serving simple static DNSSEC secured zones.

4.3 PowerDNS

PowerDNS[5] is mainly developed by PowerDNS.COM. It started as a closed source project and in 2002, the source codes were subsequently opened. It provides functionality of both authoritative and recursive name servers and provides lot of flexibility due to various storage backends. It was designed for name servers operating very large amounts of zones. Unfortunately, the documentation is spares and often missing important bits.

Enabling DNSSEC in PowerDNS is quite easy, signing is performed within the server and no third party utilities are required. However, not all backends are supported. Basically, DNSSEC can be operated in three modes:

[3] https://www.nlnetlabs.nl/projects/nsd/

[4] https://www.nlnetlabs.nl/projects/ldns/

[5] https://www.powerdns.com/

- **Serving pre-signed zone**

  This is the basic mode, when PowerDNS provides only responses from pre-signed zone loaded into the backend, or retrieved from a zone transfer.

- **Front signing**

  In this mode, all DNSSEC related records are synthesized on the fly and cached in the memory. The database contains only unsigned RRs. If NSEC3 is used for authenticated denial, the database might also store hashed name with each domain name for better performance.

  Front signing is also applicable for zones retrieved from another master server.

- **BIND-mode operation**

  PowerDNS implements a backend, which allows serving of zones from BIND compatible configuration files. Originally, the backend was started as a demonstration of versatility of PowerDNS, but later gained in importance. As the backend itself does not support a storage of metadata, this mode provides separate SQLite in-file storage for key the material.

PowerDNS comes with a separate tool for DNSSEC control, called *pdnssec*. To enable DNSSEC, just one command has to be issued:

```
$ pdnssec secure-zone example.com
```

One KSK and one ZSK are generated and immediately set as active. NSEC is used for the authenticated denial. The active keys and DS records can be displayed with:

```
$ pdnssec show-zone example.com
Zone is not presigned
Zone has NSEC semantics
keys:
ID = 1 (KSK), tag = 58474, algo = 8, bits = 2048    Active: 1 ( RSASHA256 )
KSK DNSKEY = example.com IN DNSKEY 257 3 8 AwEAAcybi1SSIWCNvUOiKeu1rzB42...
DS = example.com IN DS 58474 8 1 ec5e2d5b0901f61465299... ; ( SHA1 digest )
DS = example.com IN DS 58474 8 2 e26fcd4a97e739d8e56... ; ( SHA256 digest )
DS = example.com IN DS 58474 8 4 8327aa11d305e197e1... ; ( SHA-384 digest )
ID = 2 (ZSK), tag = 12133, algo = 8, bits = 1024    Active: 1 ( RSASHA256 )
```

To enable NSEC3, following commands are required:

```
$ pdnssec set-nsec example.com '1 0 10 C01DCAFE'
$ pdnssec rectify-zone example.com
```

The first command switches authenticated denial to NSEC3. The parameters for hashing are not required and the server will choose them randomly if not specified. The second command is needed by some backends only, and updates the ordering of hashed domain names.

The *pdnssec* command provides a bunch of subcommands to manage the keys, but no timing metadata nor hardware modules are supported:

- **generate-zone-key** generates a new private key in BIND format.
- **add-zone-key** generates a new private key and imports it into a key database.
- **import-zone-key** imports an existing key in BIND format into a key database.
- **activate-zone-key** and **deactivate-zone-key** activate or deactivate a key.
- **export-zone-dnskey** and **export-zone-key** export a key pair in a BIND format.
- **remove-zone-key** deletes a private key from a key database.

DNSSEC in PowerDNS can be enabled very easily and no third party utilities are required. However all key maintenance operations have to be performed manually and also the configuration options are very limited. Only the parameters of automatically generated keys for newly secured zones can be set. Signature lifetime is harcoded — all synthesized RRSIGs have inception time at most a week in the past and expire at least two weeks in the future (the turnover happens always on Thursday midnight UTC, because POSIX time starts on Thursday midnight).

4.4 OpenDNSSEC

OpenDNSSEC[6] is a project developed in collaboration of subjects: .SE (The Internet Infrastructure Foundation), NLnet Labs, Nominet, Kirei, SURFnet, SIDN, John Dickinson, and many others. Unlike the previously described solutions, OpenDNSSEC is not a name server, but a tool for handling the entire process of zone signing automatically, including key management and a timing of operations. Only a few operations must be performed by the zone operator.

The main idea is to separate zone signing from the name server. OpenDNSSEC takes unsigned zones and provides signed zones to a name server running aside, which then serves the signed zones. Any security aware name server can be used for this purpose.

The solution is not key-centric, but policy-centric. OpenDNSSEC allows definition of multiple sets of rules, which are assigned to individual zones. The set of rules is called Key and Signing Policy (KASP) and includes:

- **Signing parameters**

  Allows configuration of the signature refresh interval, the validity period, and the expiration time jitter (may split signer engine load for large zones). Also the inception time offset can be added to eliminate a possible time skew on resolvers.

- **Authenticated denial parameters**

  Switches between NSEC and NSEC3. For NSEC3, the hash algorithm, the iteration count, the opt-out flag, and the salt length can be specified. The salt cannot be user-defined, but is generated randomly at specified intervals.

- **General key parameters**

  Sets TTL of DNSKEY records, the safety interval for key DNSKEY changes, and the interval of old keys purging. Also, the keys can be generated separately for each zone or can be shared between all zones using the KASP.

[6] https://www.opendnssec.org/

- **KSK and ZSK parameters**

  Specifies the used algorithm, the key bit length, the key rollover interval, and the location of private key storage.

- **Zone information**

  Zone information sets the data propagation delay from master to slave servers, which must be taken into account during key rollovers. Also the SOA TTL and the serial update policy must be specified.

- **Parent zone information**

  In order to provide a safe ZSK rollover, the propagation delay of the DS record to the parent zone must be specified, and also the TTLs of the DS record and the parent SOA.

OpenDNSSEC operates in two separate cooperating processes:

- **KASP enforcer**

  This process is responsible for the key management. It generates new private keys in the key storage, controls the key backup, association of the zones with policies, keeps track of used keys and manages key rollovers by choosing current signing keys.

- **Signer engine**

  This process performs the zone signing itself. It reuses old unexpired signatures, creates new, maintains the NSEC or NSEC3 chain, and updates the SOA record according to the KASP.

The signer engine uses a generalised interface for zone retrieval of unsigned zones and distribution of signed zones, called adapters. OpenDNSSEC currently provides adapters for static master files, full zone transfers (AXFR), and incremental zone transfers (IXFR).

The simplest configuration (with master files) reads the unsigned zones from one directory and stores them in a second directory. When the signing is complete, the signer can execute an arbitrary command, which may tell the name server to reload the zone file.

Zone transfer may be used when the zone is dynamically updated using DDNS. It requires a hidden master server, which retrieves the DDNS updates and provides the unsigned zone to OpenDNSSEC via the zone transfer. The signed zone is then passed to a real master server. Unfortunately, the delays between DDNS update and change publication can be longer than with the solution performing the signing within the name server.

OpenDNSSEC can be used to build a very robust signing solution due to the separation of all components. This allows to include additional components into the process, like automatic auditing. OpenDNSSEC also supports hardware security modules on a very high level. In fact, only PKCS #11 devices can be used as a key storage. The SoftHSM software token, which was created as a part of the project, can be used for small deployments and in a testing environment.

An intervention of the zone administrator is required for ZSK rollovers, when the DS record publication must be performed in the parent zone. Additionally, OpenDNSSEC can monitor backups of private keys and the KASP can be configured not to use any key, which was not backed up. The backup is another action, which must be performed manually. Otherwise, OpenDNSSEC can operate autonomously.

4.5 Knot DNS

Knot DNS[7] is an authoritative-only name server developed by CZ.NIC Labs. It is designed for high-performance and non-stop operation, even for very large TLD zones. Support for DNSSEC pre-signed zones was included in the first beta release in 2011. Experimental support for automated signing is included since version 1.4, which was released in January 2014.

Server configuration for pre-signed zones is simple:

```
zones {
    example.com {
        file "/var/lib/knot/example.com.signed";
    }
}
```

For automated DNSSEC signing, the signing keys have to be generated and signing must be enabled in the configuration file. Knot DNS does not provide utilities for key generating yet, but supports keys in the BIND format including the timing metadata. Therefore *dnssec-keygen* (and optionally *dnssec-settime*) from BIND project can be used. All keys for a zone must be placed in one directory. Then, `dnssec-enable` and `dnssec-keydir` options in configuration file must be set properly:

```
zones {
    example.com {
        file "/var/lib/knot/example.com";
        dnssec-keydir "/var/lib/knot/keys";
        dnssec-enable on;
    }
}
```

When the zone is loaded into the server, the DNSSEC related RRs in the zone are updated. According to timing metadata in key files, DNSKEYs are published and RRSIGs updated or generated. If the zone contains NSEC3PARAM RR, NSEC3 is used instead of NSEC. The NSEC3 opt-out is not supported and thus all insecure delegations are always signed.

Automatic signatures also apply to DDNS updates. However, automatic signing cannot be used with zones retrieved by a zone transfer.

4.6 Yadifa

YADIFA[8] is a lightweight authoritative name server by EURid. It supports DNSSEC only via pre-signed zones loaded from master files. Some configuration options for automatic signatures are already accepted, but have no effect. YADIFA does not provide any signing utilities, therefore utilities from BIND or ldns must be used.

[7] https://www.knot-dns.cz/

[8] http://www.yadifa.eu/

	BIND	Knot DNS	PowerDNS	NSD	Yadifa	OpenDNSSEC
Serving signed zone	●	●	●	●	●	–
Tools for signing	●	–	–	●	–	–
Key files support	●	●	●	●	–	–
PKCS #11 support	●	–	–	–	–	●
Automatic signing	●	●	●	–	–	●
DDNS signing	●	●	●	–	–	–
Key management	–	–	–	–	–	●

Table 4.1: The overview of available DNSSEC features of the compared software.

No special configuration options are needed for serving a pre-signed zone:

```
<zone>
    domain example.com
    type master
    file example.com.db.signed
</zone>
```

4.7 Existing Implementations Summary

The Table 4.7 visualizes available DNSSEC features in the analyzed solutions. All solutions in the table are authoritative server implementations, except for OpenDNSSEC. OpenDNSSEC uses different approach and works as a DNSSEC signing service.

All of the servers support DNSSEC at least in the simplest form, which is a serving of pre-signed zone. BIND and NSD also provide tools for creating such a zone. BIND, Knot DNS, and Power DNS additionally support some mechanism for automatic zone signing. In case of BIND and Knot DNS, the server creates all DNSSEC RRs before serving the zone. In case of PowerDNS, the DNSSEC RRs are synthesised at the time of answering. At the moment, no described server supports automated key management.

Chapter 5

DNSSEC Library Design and Implementation

This chapter describes use cases and design goals for the new DNSSEC library and auxiliary tools. The design goals are based on the requirements for the use of the library in the Knot DNS authoritative name server, and also on the results of existing solution analysis. The chapter also covers some implementation details of the library, namely the used cryptographic backend, conversion of key and signature formats, and existing abstraction layers. In the end, the project structure and testing is explained.

Most available open-source solutions for DNSSEC use manually generated signing keys. Security policy regarding the rollover of these keys is mostly enforced by hand as well. In a better case, timing metadata are attached to the keys and the process is performed semi-automatically. Although utilities for verification of the settings are available, the whole process is still prone to user errors. The verification utilities may also miss some special cases, like algorithm rotation.

Except for custom scripts, the only solution for automatic and safe key management is OpenDNSSEC with Key And Signature Policy (KASP). The KASP steers the key exchange according to the policy itself, which is a much intuitive abstraction, than the timestamps in key files. On the other hand, OpenDNSSEC is fragile in corner cases, when the zone operator needs to perform some operation entirely by hand.

The library, which is the objective of this work, aims to create a basis allowing to build solutions providing the same comfort as the KASP in OpenDNSSEC, while preserving the flexibility of timing metadata in BIND key files. Because of this, KASP concept will be adapted but modified to allow overriding or suppressing of automatically performed actions.

The library will be initially used with Knot DNS. Therefore, the included functionality will primarily provide functions covering DNSSEC operations performed by authoritative name servers. The support for procedures performed only by resolvers is not a priority and may be a subject of future extensions of the library. With respect to possible usage in other name servers, the library must provide such a level of abstraction, which will not create restrictions upon the internal structures used by the name server.

The new library will be accompanied with a set of tools, which will facilitate the management of a KASP and manipulate keys, and tools generally useful for troubleshooting DNSSEC. Also simple alternatives for common DNSSEC tools (e.g., *dnssec-keygen*, *dnssec-signzone*) may be added to demonstrate the functionality of the library.

5.1 Use Cases for the Library

To define the basic outlines of the library, the lifetime of a secure zone can be taken as a starting point. Operations performed with the DNSSEC RR will provide some hints.

1. **Define KASP for the zone.** The prerequisite for securing a zone is the definition of the KASP for that zone. The policy defines the signing scheme, the key parameters, and the signature parameters.
2. **Create the signing keys.** The next step in securing the zone is the generating of an initial key set as specified in the KASP. In case of moving from one DNSSEC solution to another, the existing keys might be imported instead. The keys are stored in a key store.
3. **Establish trusted chain.** To establish a trusted chain with a parent zone, DNSKEY records must be included in the zone and corresponding DS records must be published in the parent zone. These records are derived from the public keys. The published DS records may also include standby keys.
4. **Create records for authenticated denial.** When DNSKEY records are in place, the chain of records for authenticated denial must be added into the zone. If the hashed authenticate denial is used, the salt for hashing must be generated before that.
5. **Sign the zone.** All authoritative RR sets in the zone must be signed. The process covers identification of authoritative RR sets, conversion of these sets into the canonical format, selection of private keys, creating the signatures, and adding RRSIG records into the zone. The selection of the keys is dependent on the state of key and on the signed RR type.
6. **Maintain the zone signatures.** Expiring signatures in the zone must be refreshed in time. Also new or updated RRs must be created. Changes of the zone structure influence the NSEC or NSEC3 chain, which must reflect the changes as well.
7. **Maintain the keys.** The KASP defines a key rollover interval. Prior to the key rollover, the server must request a new key to be generated. The rollover usually consists of multiple steps, which are split by delays required for data propagation and a cache expiration. The new keys may be backed up, the old keys are usually purged.

5.2 Design Goals

- **Key storage**

  The library will provide interface for key storage, which will be tightly bound to the KASP. The storage must provide space for public keys, private keys, and their metadata. The metadata will contain parameters used in DNSKEY records (flags and algorithm identification), timing information mapping the past and future states of the key in it's lifetime (e.g., created, published, active, retired, dead, etc.).

  The storage will also provide mechanisms for retrieval of keys by the key tag (likely non-unique identifier), by other unique identifier (e.g., cryptographic hash of public key), and the current lifetime period.

- **Key generating**

  Given the key storage, the library will provide an interface for generating of new key pairs according to given parameters into the storage.

- **Key importing and exporting**

  Library will provide interface for key pairs importing and exporting. Portable PKCS #8 format will be used for private keys [54]. For public keys, DNSKEY and DS export format is a must.

- **Private keys in PKCS #11 device**

  The storage must allow to store private keys on hardware security modules and use PKCS #11 interface to work with these keys.

- **Support for offline keys**

  The key store will support offline keys, where the private key is available only temporarily or never. Temporary availability could be the case for security hardware modules. And in some secure environments, KSKs are stored on a separate device and ZSKs are signed there. For this purpose, the library will support storing of pre-signed DNSKEY sets.

- **Support for standby keys**

  To be able to perform fast KSK rollover in case of an emergency, standby keys are often used. The DS for these keys is published in the parent zone, but the ZSK key does not appear in the zone until the rollover is initiated. The library will support this operation.

- **Signing schemes support**

  Based on the data in a key store, the library will determine, which signing scheme is used and provide hints to the name server when choosing keys to be used. Primarily, KSK—ZSK signing scheme and single type signing scheme will be supported.

- **Bit map construction for authenticate denial**

  NSEC and NSEC3 records contain a bit map field encoding RR types present at the domain name in the zone. The library will implement an interface for comfortable construction of these maps. As the rules for inclusion of individual types are slightly different for NSEC and NSEC3 at delegation points, the checks will be included.

- **NSEC3 hash operations**

  For the purpose of NSEC3, the library interface will include random salt generating and domain name to hash conversion functions.

- **Low level signing and verification**

  The signatures in DNSSEC may use different encoding than specified in X.509 [6, 19]. The X.509 format is used by most cryptographic libraries. The library will provide transparent interface for encoding and validating arbitrary data in the format used by DNSSEC.

- **Resource record set signing and verification**

  As server implementations use specialized structures for storage of zone data, the abstraction will provide signing of raw data and RRs in the wire format. Organizing the process of signing within the zone will remain the responsibility of the name server.

- **Auxiliary functions for signing and verification**

  The keys used to sign new, updated, or expired RR sets depend on the type of the records, their location, and on used signing scheme. The library will provide interface for choosing the correct keys based on these circumstances.

  The comparision of time expressions in signatures requires serial number arithmetic [24], because short data types are used [4]. Helper functions performing this arithmetic will be included as well.

- **Key and Signature Policy**

  The policy must include parameters for signing, authenticated denial, key lifetime, and key rollover. OpenDNSSEC also defines other parameters related to delays of data propagation to slave servers and the parent zone. These parameters can be probably determined more precisely by the name server using the library, however the utilities working with the KASP outside the server process would not have these information. Consequently, these parameters will be optionally included in the KASP as well and may be updated by the name server.

  Also, the KASP storage will provide space for holding custom parameters related to the policy. The name server could use them arbitrarily. For example, the KAPS can contain zone operator e-mail address to be used to inform about emergency situations. This approach will allow to store all related information on once place and not to spread them between the KASP and the server configuration file.

- **KASP override mechanism**

  The library will provide interface to override key lifetime state enforced by the KASP, which in fact enables manual control over the key as used in BIND name server.

- **Key rollovers**

  Zone maintenance also comprises rollovers of keys, which is a process controlled by the KASP. The rollover is a sequence of changes to DNSKEY records and also RR signatures in the whole zone. The set of keys used at a given time will be determined by retrieving the keys from key storage by their lifetime state. The rollover events can happen independently on the name server operation, but need not be executed in real time. Their late execution will not cause any harm.

  The library will provide a set of functions, which will be used to handle these events. The name server will retrieve the next KASP event and it's details, perform the requested operation not soon than at the planned time, and finally mark the event as executed. At that point, the library will update key states, which may determine the next rollover events.

- **SIG(0) transactions signing and verification**

  The DNS allows transaction signing using multiple mechanisms. SIG(0) is a mechanism, which is similar to RR signing [16]. Although transaction signing is out of

DNSSEC scope, the library will provide interface for DNS messages signing and verification using SIG(0).

- **Abstraction for KASP and key storage**

  The library will provide abstraction for KASP and key storage backend. To store only data for a few zones, storage on the file system might be sufficient. In case of large amount of zones, other backends might scale better (e.g., SQL database). It will be possible to choose a key storage backend at runtime, and even to use multiple storages at a time.

  The library implementation will contain one backend for storage on the file system in a human readable and standardized format (e.g., JSON, YAML). Other backends might be added by third parties.

- **Construction of trust chain**

  While the construction of trust chain is not in a scope of authoritative name servers, the interface for construction and verification of the trust chain will be added. The interface is going to be needed by some of the accompanying utilities. And possibly will make the library more attractive to validating resolvers.

5.3 Utilities

- **KASP management utility**

  As the library provides abstraction for KASP storage, an utility will be provided to control the KASP from a command line. Generally, the utility will enable:

  - Creating, listing, and deleting policies.
  - Displaying of policy parameters, and modifying these parameters.
  - Managing of zone-to-policy mappings.
  - Exporting and importing of the policies.

- **Key management utility**

  Although the key storage is tightly bound to KASP, the key management will be provided by a separate utility. The utility will allow:

  - Listing existing keys for a zone.
  - Listing upcoming rollover events for a zone.
  - Manually generating, importing, and exporting the keys.
  - Generating keys in advance according to KASP (for standby KSK).
  - Displaying details for a key.
  - Overriding KASP by setting the key lifetime parameters.

- **Utility for DNSKEY RR sets signing on isolated device**

  If the policy requires the private KSK to be present on a machine isolated from the name server, another tool will be added and provide functionality to:

  - Export the public keys and metadata describing the timing of the upcoming rollovers.
  - Sign the key sets in the rollover by selected KSK and export them.
  - Import the signed key sets into the key storage.

- **Private key conversion utility**

  Most of the current name servers supporting DNSSEC signing adopted the BIND key format. The public key file contains DNSKEY RR in a zone master file format. The private key stores values used in the cryptographic algorithm and timing information for management of the key rollover. All data are encoded as ASCII text, which makes it both human readable and easy to parse by custom scripts.

  The BIND keys fulfill their purpose when the keys are rolled manually. The public key can just be copied into the zone and the timestamps in private key can be modified with a text editor. However, no manual modifications make sense in case of the automatic rollover managed by a KASP. Also, the BIND key format is not supported by cryptographic libraries and complicated conversion must be performed.

  The library will store the private keys in PKCS #8 format, which is a standard format for storing private key information and is supported by most cryptographic libraries. In addition, the format allows protection of the key material with encryption and a pass phrase. To cooperate with other DNSSEC components capable using only BIND key format, a conversion utility will be distributed with the library.

- **DNSSEC hash calculation utility**

  DNSSEC uses various hashes at multiple places. The utility will allow:

  - Compute key tag from DNSKEY record.
  - Compute DS record from DNSKEY record.
  - Compute NSEC3 hash from given domain name and hash parameters.

- **Signature chain testing utility**

  For testing of DNSSEC setup, a simple resolving utility will be added. This utility will display the chain of trust and a sequence of DS and DNSKEY records used to validate the RRSIG.

- **Zone signing and verification**

  Utilities similar to *dnssec-signzone* and *dnssec-verify* will be added, which will use the KASP as a source for signing information. The utility will probably depend on external zone parser library (most likely *zscanner* from Knot DNS).

5.4 Cryptographic Operations

An essential part of the DNSSEC specification relies on cryptographic operations. As there are huge differences in interfaces of various cryptographic libraries, a selection of the backend library had to be performed at an early stage of the design. The choice also determined some mechanisms for manipulation with key material and also conversions between supported encoding formats.

5.4.1 Selection of a Cryptographic Library

The initial specification for the library contained a layer, which would make usage of different cryptographic libraries possible. However, this requirement was identified to be superfluous. An additional abstraction layer would complicate the implementation needlessly with just a little added value. Therefore, only a one cryptographic backend is supported. Following cryptographic libraries were considered:

- **OpenSSL**

  OpenSSL[1] is a library used by most existing open-source DNSSEC software. It provides functions for general purpose cryptography and high performance. Unfortunately, the library interface is relatively unstable, which causes problems when compiling the software on multiple systems with different versions of the library. In addition, the documentation is often incomplete and the source code itself is difficult to read.

  Also, OpenSSL does not support cryptographic tokens accessible via PKCS #11 interface. The support for PKCS #11 can be enabled using external engine *engine_pkcs11*[2] developed as a part of the OpenSC project. However the external engine is complicated to use and I was not able to make it work with *SoftHSM* software token, *ePass 2000* hardware token, nor *Sun SCA6000* cryptographic accelerator.

- **Mozilla Network Security Services**

  The Mozilla NSS[3] library was initially designed to be used in the Mozilla Suite. From this reason, a lot of the internal operations work only with a global context (e.g., state of PKCS #11 tokens). Therefore multiple uses of the library within one application cannot be isolated very well. Similarly to OpenSSL, the documentation of the library is very poor.

  The support for PKCS #11 by the Mozilla NSS is very good. However, even the low level public key operations require an X.509 certificate, which are not used in DNSSEC. That would make the library very problematic to use, because an artificial certificate would have to be created.

- **GnuTLS**

  The GnuTLS[4] library provides a very good alternative for the previously mentioned libraries. It offers an abstract interface for performing public key operations using both software keys and hardware devices, including the PKCS #11 ones. The documentation of the library is well organized and often contains theory basics with references to literature. In addition, the API is illustrated with a lot of examples.

At the moment, DNSSEC allows signing with the RSA, DSA, ECDSA, and GOST algorithms. All these algorithms, except of GOST, are supported by the referenced libraries. The GOST algorithm is usually supported only through an external patch. From this point of view, the libraries provide equivalent functionality.

[1] https://www.openssl.org

[2] https://www.opensc-project.org/opensc/wiki/engine_pkcs11

[3] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS

[4] http://gnutls.org

In the end, the GnuTLS library was chosen as a best solution for the designed library. The main reason is a good documentation, satisfactory support for PKCS #11, and last but not least to increase the diversification between available open-source DNSSEC solutions.

5.4.2 Key and Signatures Encoding

Nowadays, most Internet protocols use Transport Layer Security (TLS) to provide communication security between the communicating parties. The TLS protocol uses X.509 format to encode certificates, keys, and signatures [12, 15]. For this reason, the cryptographic libraries (including GnuTLS) primarily use the X.509 format when managing keys and performing signing operations.

DNSSEC adapts a custom format for encoding of keys and signatures. Consequently, various conversions need to be performed within the DNSSEC library.

Format of DNSSEC Private Keys

The DNSSEC specification does not define a specific format for private keys since these keys are not directly used in the DNS protocol. Historically, the format was appointed by BIND as a first implementation of DNSSEC. The BIND private key contains algorithm parameters encoded in Base64.

An example of a BIND private key follows:

```
Private-key-format: v1.3
Algorithm: 3 (DSA)
Prime(p): t+xj7VAqPe5ubujgchnk3Y2JqReYyv3O77iLaXxue1NCf9gKEM7dnlRMTYg8VA...
Subprime(q): /BlDeI7HSoKKOOTPKNTyxVgCk/M=
Base(g): cL0aNeGPPJsTq7GGy147e2QAT6rW7uvUKEoXUFk6Ddo8p0BkigBRHku9b62ecDi...
Private_value(x): R+ErDDVccNO1pzbaxIQP2MTuv5U=
Public_value(y): g0DdN1aNHXgytZWGqhHavieY0z269sYDy2edp0HvV0ks+vQEQeAyUav...
Created: 20140425184418
Publish: 20140425184418
Activate: 20140425184418
```

The format is easily processable by user scripts and allows inclusion of additional fields, e.g. key usage timing metadata. However, the key cannot be used in other applications without conversion to standardized format. In addition, for ECDSA keys, a corresponding public key cannot be derived from the private key file in the BIND format.

The designed DNSSEC library uses X.509 PEM format to store private keys. This format is processable by most cryptographic libraries and tools supplied with these libraries. It cannot keep additional data — in this case, timing metadata and an identification of the cryptographic hash function. However, this information can be bound to the zone configuration. As a result, the key can be reused by multiple zones and the library need not to distinguish between different types of key stores (e.g., the PKCS #11 allows to store custom attributes alongside a private key, but the attributes are immutable).

The private key in X.509 PEM format, matching with the previously shown BIND key, looks like:

```
-----BEGIN PRIVATE KEY-----
MIHGAgEAMIGoBgcqhkjOOAQBMIGcAkEAt+xj7VAqPe5ubujgchnk3Y2JqReYyv3O
77iLaXxue1NCf9gKEM7dnlRMTYg8VAuCuZpsHkteGDDjxJqYZMN7TwIVAPwZQ3iO
x0qCijjkzyjU8sVYApPzAkBwvRo14Y88mxOrsYbLXjt7ZABPqtbu69QoShdQWToN
2jynQGSKAFEeS71vrZ5wOLwRtwQzm9OuNSzzQ+G7d32/BBYCFEfhKww1XHDTtac2
2sSED9jE7r+V
-----END PRIVATE KEY-----
```

Format of DNSSEC Public Keys

Public keys in DNSSEC appear as DNSKEY RRs, which were already described in section 3.3. The key material is stored within the Public key field of the DNSKEY RR and it's format depends on the used algorithm [19, 20, 31]. Basically, public key attributes are just encoded as byte strings of defined width and padding. The GnuTLS library supports public key raw attributes import and export, and the designed DNSSEC library utilizes this interface to read and write the DNSKEY RRs.

The BIND stores public keys in a zone master file format. The public key file for the example private key would contain:

```
example.com. IN DNSKEY 256 3 3 APwZQ3iOx0qCijjkzyjU8sVYApPzt+xj7VAqPe5ub...
```

The key can be included into the zone simply by adding the file content into the zone master file. This is a significant advantage, when the zone is signed manually using the command line tools. However, this property is no longer needed if the keys are maintained automatically. As the designed DNSSEC library aims to perform automated key maintenance, the ability to manipulate public keys in zone master file format is not included.

Format of DNSSEC Signatures

The DNSSEC signature is held within the Signature field of RRSIG RR. Similarly to the public keys, the format of the signature depends on used algorithm:

- For the **RSA algorithm**, the signature is represented by one value, which is encoded the same way as in X.509 certificates [20, 35]. This was intended to make standard cryptographic libraries easier to use.

- For the **DSA and ECDSA algorithm**, a conversion is required [6, 19, 31]. In this case, the signature consists of two values usually called r and s. In the X.509 specification, the storage of these values is described by the *Dss-sig-value* (and identical *Ecdsa-sig-value*) ASN.1 structure. DNSSEC encodes the signature values as byte strings on a fixed width. For future extensions, the DSA signature is supplemented with a one-byte value t, copied from the public key.

The X.509 signatures are encoded using Distinguished Encoding Rules (DER) standard. Internally the GnuTLS library uses *Libtasn1*[5] library to perform the encoding. Unfortunately, no public interface is provided to access the raw signature parameters. In order to write and validate the DSA and ECDSA signatures in DNSSEC, this functionality was implemented as a part of the DNSSEC library.

[5] https://www.gnu.org/software/libtasn1/

The Libtasn1 library includes a utility, which converts a textual ASN.1 definition into a C source code. The resulting code can can be used to initialize the Libtasn1 parser or encoder. This is a very comfortable way to access complex binary structures, however the structures used in the DSA and ECDSA signatures are fairly plain. For this reason, the Libtasn1 library was not use and a simple one-purpose parser for *Dss-sig-value* was implemented as a part of the DNSSEC library.

Key Identifiers

There is no standardized way to identify a cryptographic key. Usually for this purpose, cryptographic libraries adapt hashing of some part of the public key and identify the key using the result. In the designed DNSSEC library, following key identifiers can be used:

- **Key Tag**

  DNSSEC defines a simple 16-bit hash value called key tag, which is calculated from the DNSKEY RRs and appears in the RRSIG and DS RRs [4]. The key tag is not a unique identifier of the public key. It is used only to limit the possible candidate keys during signature validation and DNSSEC trust chain building.

- **Key ID**

  In order to identify a key uniquely, the DNSSEC library introduces a key ID. The key ID is a SHA-1 hash value computed from the public key. In this context, the public key means a *Subject Public Key Info* structure in the X.509 specification, encoded in the binary DER format. The key ID can be used to select a key unambiguously during the signing. As the value is computed only from a public key, the identifier is independent on DNSKEY flags or hashing algorithm changes.

5.5 Signing State Persistence

As the DNS system is decentralized, an authoritative server has only a limited knowledge about the data maintained by other servers. And generally, there is no way to track changes made in a specific zone.

The DNSSEC records in a zone are bound together at the level of the zone (RRSIG and DNSKEY RRs) and at the zone cut level (DNSKEY and DS RRs). Although the records are semantically bound, resolvers handle all records individually. This fact makes some DNSSEC operation complicated, especially key rollovers. From this reason, the DNSSEC library has to keep some additional information for each zone and consult that information before invoking any change to an authoritative data.

5.5.1 Signing State Persistence Model

In order to preserve the DNSSEC validation chain in all situations, the key states has to be considered from the point of view of a caching resolver and the longest possible caching periods have to be taken into account. This expects knowledge of data propagation delays, exact times of DNSKEY RRs publication, and TTL times of related DNSSEC RRs.

Figure 5.1 shows a simplified ER diagram with entities involved in storing the signature policy and state of the zone signing. The meaning of the individual entity types will be explained.

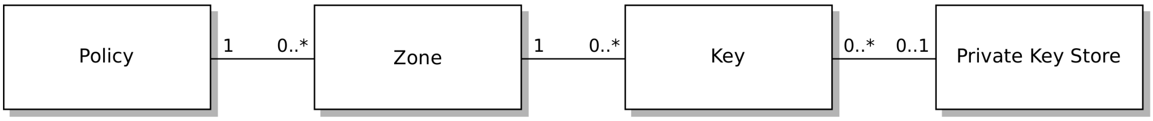

Figure 5.1: Simplified ER diagram of entities involved in key and signature policy persistence.

The Policy Entity Type

The policy is a definition of zone signing configuration. The configuration covers all the information, which need not be altered, when the zone content changes. Therefore the policy can be shared by multiple zones. The policy consists of:

- Key parameters:
  - TTL of DNSKEY records;
  - algorithm and private key length for ZSK and KSK; and
  - lifetime for ZSK and KSK (how often the keys are rolled over).
- RRSIG parameters:
  - validity time;
  - refresh time (interval prior expiration, when the signature renewed); and
  - expiration time jitter (to avoid mass expiration of zone records).
- NSEC3 parameters:
  - option to enable NSEC3 (otherwise NSEC is expected);
  - option to enable of Opt-out flag;
  - salt length and hash function iteration count; and
  - salt rollover time.
- Parent zone parameters:
  - SOA record TTL and minimal TTL; and
  - DS record TTL.
- Data propagation delays:
  - master to slaves propagation delay;
  - inception offset (to reduce issues with time synchronization); and
  - DNSKEY publish and retire safety interval (prolongs each key rollover step).

The Zone Entity Type

The Zone entity type represents a state of a signing for a particular zone. The state is determined by a policy and a set of keys with their timing metadata.

The Key Entity Type

The Key entity type represents a key instance used with the zone, and includes:

- **DNSKEY RDATA**

  In order to build a DNSKEY RR to be inserted into the zone, the Key entity stores a complete DNSKEY RDATA. The zone name is used as a domain name for the RR, the TTL value is taken from the policy specification.

- **Key timing attributes**

  The designed DNSSEC library does not use a fixed number of states to describe a key lifetime (unlike OpenDNSSEC), but uses various timing attributes (like BIND). The timing attributes track the usage of the key in the zone, based on which a current state of a key is resolved. The attributes are:

  - Publish — The time when the DNSKEY RR is included into the zone.
  - Activate — The time the key is started to be used for signing.
  - Retire — The time the key is no longer used to create signatures.
  - Remove — The time when the DNSKEY RR is removed from the zone.

  And the values of the attributes are limited by a few constraints:

  $$\text{Publish} < \text{Remove} \quad \wedge \quad \text{Activate} < \text{Retire}$$

  There are no constraints on the Publish versus Activate attributes. This allows modeling of exceptional states, where an RRSIG RR for one key appears in the zone before the DNSKEY RR for the same key. This particular situation is required for key algorithm rollover [39], and unfortunately cannot be performed by OpenDNSSEC.

  Also, the use of the timing attributes instead of fixed states eases a transition from the BIND software.

- **Reference to a private key store**

  The reference to a private key store has to be specified in order to use the key to create new signatures. In special cases, this might not be necessary (e.g., when the key is retired after migration from other DNSSEC signing solution).

The Private Key Store Entity Type

The type represents an instance of a private key store. The private keys are stored apart from the public keys for several reasons:

- An isolated storage for private keys increases the security. The private key is loaded only if required for signing.
- The keys can be shared by multiple zones, which is a practice of some DNS hosting providers.
- The private keys are immutable and can be stored on immutable storage.
- The KSK can be stored offline, and attached only when the key rotation happens.

5.5.2 Signing State Store Abstraction

The library encompasses two parallel abstraction layers, which allow implementation of different storage backends:

- a layer for signing state storage; and
- a layer for private key storage.

These two abstraction layers have to be implemented separately because of a support for cryptographic tokens. The tokens do not serve only as a storage for key material, but perform certain cryptographic operations individually. This method makes a unified access to a key storage impossible (without exposing the underlaying cryptographic layer).

Signing State Store Abstraction Layer

The signing state store abstraction layer corresponds with the ER diagram in Figure 5.1, excluding the Private key store entity type. The meaning of the individual entities was already described.

Currently, the interface is incomplete. Only a default implementation for the signing state store is exposed in the public interface of the library, the abstraction layer is available only internally and is subject to change:

- KASP — The private key store specification is not included. As the key store abstraction layer is separate, a method of instantiation of the storage is not decided yet. This is especially problematic for non-default implementation.
- Policy — The structures for specification of signing policy are in a state of a draft and are not used by other entities yet.
- Zone and Key – The structures are generally complete.

The default implementation of the signing state store uses textual files, which are retained on the file system as usual. The content of the files is written in YAML (*YAML Ain't Markup Language*[6]), which is a data serialization format with a human readable notation. There are a lot of YAML libraries, which allow to process the format in many popular languages. Thus the administrators can comfortably process the files both manually and automatically by custom programs or scripts.

An example of signing state file for zone `example.com` may contain:

```
policy: default
keystore: default
keys:
  - id: 788e1e2d37ab359899735349e38442cc8d038795
    ksk: false
    algorithm: 3
    public_key: APwZQ3iOx0qCijjkzyjU8sVYApPzt+xj7VAqPe5ubujgchnk3Y2JqReY...
    publish: 2014-04-25 18:44:18
    active: 2014-04-25 18:44:18
```

[6] http://www.yaml.org/

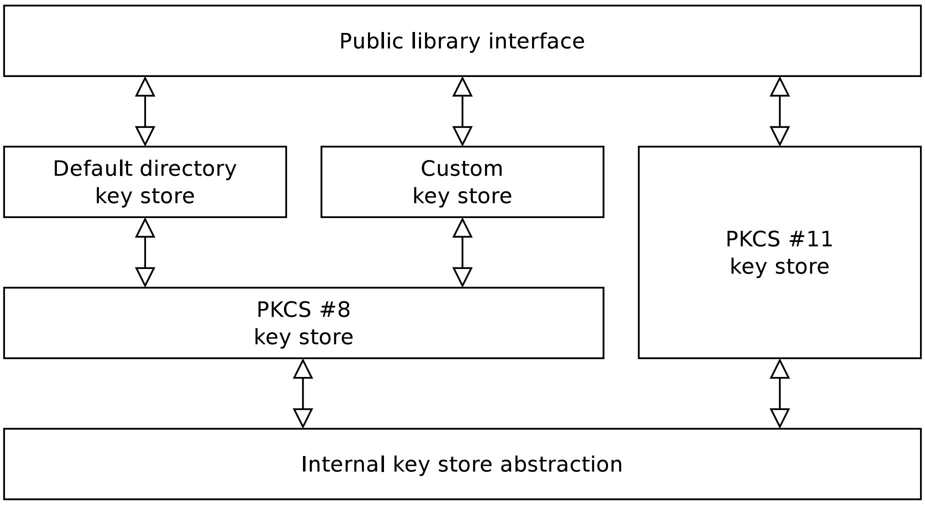

Figure 5.2: Key store abstraction API in the DNSSEC library.

Any implementation for the zone state store must be able to reconstruct he Zone and Key entities. The default implementation stores all necessary fields within the zone state file, except the name of the zone. The name of the zone is taken from the filename (in this case `zone_example.com.yaml`). In the example, the lines one and two refer to a policy and an implicit key store, the line three introduces a list of key instances. The DNSKEY RDATA for each key are reconstructed from the *ksk*, *algorithm*, and *public_key* fields. The *publish* and *active* fields specify a current state of the key. The *id* field contains a key ID, which can be computed from the public key and therefore represents a redundant information. However, the field is handy for the administrator to ease a lookup of the private key without additional tools.

Private Key Store Abstraction Layer

The private key store abstraction layer is shown in Figure 5.2. Basically, two methods for accessing the keys are distinguished:

- **PKCS #8 key store** (software key)

  In this case, the key store provides a key material encoded in the unencrypted PEM format. The key material is passed to the GnuTLS library, which directly performs the cryptographic operation.

  For this purpose, the DNSSEC library contains an additional abstraction layer to allow to use a custom key store. Essentially, an implementation for key store initialization and key material access has to be provided. For convenient use of the library, a default key store implementation is available, which stores the private keys in a directory and uses the key identifiers as filenames.

- The **PKCS #11 key store** (cryptographic token)

  Cryptographic tokens can be used with the DNSSEC library via the PKCS #11 interface. The access to the token is accomplished using a dynamic library, which has to be provided by the vendor of the individual token and which implements the interface defined by PKCS #11. The key store is initialized with a configuration string identifying the dynamic library, token, and slot.

Usually, the cryptographic operation with the private key is carried out on the token and the application just picks up the result. This approach increases the security of the private key, because the key material should not leak into the application. Some hardware tokens also contain specialized acceleration chips, which offer better encryption performance than software computation of the signatures.

5.6 Project Structure and Testing

The DNSSEC library was developed as a separate project since the beginning. However, as the library is expected to be used by the Knot DNS server primarily, the source code of the library is maintained in the Knot DNS code repository. This is intended to ease API changes in the DNSSEC library, until the library is feature complete and stable enough.

A snapshot of the repository with the sources is part of the appendix B. The code of the DNSSEC library is located in the directory `dnssec` within the repository and integrates into Knot DNS build process, which uses GNU Autotools[7] build system. The DVD also contains a virtual machine image with prepared environment to compile and test the Knot DNS and the DNSSEC library.

Library Dependencies

The DNSSEC library depends on:

- the GnuTLS library for cryptographic operations;
- the Nettle[8] library for Base64 conversion;
- modified Circular Linked Lists from the LibUCW[9] library;
- the LibYAML[10] library for YAML parsing and writing; and
- the C TAP Harness[11] library for unit testing.

Source Code Organization

Internally, the source code is split into the following directories:

- *lib* — the resulting *libdnssec* library;
- *shared* — static library used internally by *libdnssec* and utilities;
- *util* — library utilities; and
- *tests* — unit tests for the internal library and *libdnssec*.

[7] http://www.gnu.org/software/software.en.html

[8] http://www.lysator.liu.se/~nisse/nettle/nettle.html

[9] http://www.ucw.cz/libucw/

[10] http://pyyaml.org/wiki/LibYAML

[11] http://www.eyrie.org/~eagle/software/c-tap-harness/

Documentation

The documentation of the library is generated using Doxygen[12]. Both internal and public data structures and functions are documented. Also, code samples with a description are provided. The source code of unit tests can be considered as an additional source of examples, if the expected behavior is unclear.

Unit Testing

For execution of the unit tests, the C TAP Harness library was adopted, as the same library is already used by the Knot DNS software. The unit test cover most of the internal and public interface of the library.

Functional testing

The functional testing can be taken into account in case of the designed utilities and the library consumers. As the utilities are largely incomplete at the moment, they are not tested at all. However, Knot DNS has a quite large suite of functional tests, which was used to test the library from the point of the library consumer. The functional tests of Knot DNS were supplemented by manual testing, which currently could not be performed automatically due to the incomplete utilities.

[12] http://www.stack.nl/~dimitri/doxygen/

Chapter 6

Conclusion

The DNSSEC designers intended to create a solution, which is backward compatible with secure unaware components of the DNS and which do not increase a message processing cost on a server side excessively. As a result, a DNS zone is secured simply by adding specialized resource records into the zone, which are understood and handled by DNSSEC aware components.

Creation of the DNSSEC records is a straightforward process defined by the DNSSEC specifications. However, complications may appear because of the architecture of the DNS. Any change to DNSSEC records in a zone must not invalidate existing data, which could be present in remote caches. Therefore, the data propagation delays and cache lifetime has to be always taken into account.

The first DNSSEC implementations required a zone operator to perform a lot of actions manually. With current utilities, nearly all of the actions can be automated. But still in most cases, the key management remains under a manual control of a zone operator. All authoritative DNS servers researched in this thesis accepted this concept, where the keys determine the zone signing policy.

A new concept was introduced in the OpenDNSSEC project. The project does not implement a name server, but a service to be run independently and to prepare secured zones to be served by any DNSSEC-enabled name server. OpenDNSSEC uses Key And Signature Policy (KASP) to define the rules for the zone signing and in addition for the key management. KASP specifies variables, which are a necessary precondition for a fully automated DNSSEC signing.

In this thesis, a new DNSSEC library and auxiliary utilities were designed. The solution is inspired by existing open-source DNSSEC software and focuses on high-level of automation and prevention of mistakes. The key features of the new library include:

- automatic key management based on the KASP concept;
- support both for software encryption and hardware security modules;
- abstraction for custom implementation of the key and zone signing state storage; and
- error-resilient interface for zone signing and signing policy enforcement.

At the moment, the implementation covers the low-level signing interface, key management interface, abstraction for key and signing state storage, and a base code for the policy definition. The designed utilities are rather prototypes.

In order to validate the functionality of the implementation, the existing DNSSEC code in the Knot DNS server was ported to the new library. The server utilizes the default implementation of the key and signing state store, the low-level DNSSEC signing interface, and the interface for TSIG transactional security. The TSIG support was added into the library aside from the original design, because it was the last feature in Knot DNS, which depended on the OpenSSL library.

Although the library provides all operations required for use in Knot DNS, the overall solution cannot be used comfortably especially due to the missing utility for key management. An implementation of this utility is therefore a top priority for future work on the library. Afterwards, the signing policy needs to be implemented, which is a key feature for the automation of high-level DNSSEC operations. The progress on the other components of the library will be arranged by the needs in the Knot DNS software.

Bibliography

[1] P. Albitz and C. Liu. *DNS and BIND*. 5th. O'Reilly, May 2006, p. 640. ISBN: 978-0-59-610057-5.

[2] R. Arends et al. *DNS Security Introduction and Requirements*. RFC 4033 (Proposed Standard). Internet Engineering Task Force, Mar. 2005. URL: http://www.ietf.org/rfc/rfc4033.txt.

[3] R. Arends et al. *Protocol Modifications for the DNS Security Extensions*. RFC 4035 (Proposed Standard). Internet Engineering Task Force, Mar. 2005. URL: http://www.ietf.org/rfc/rfc4035.txt.

[4] R. Arends et al. *Resource Records for the DNS Security Extensions*. RFC 4034 (Proposed Standard). Internet Engineering Task Force, Mar. 2005. URL: http://www.ietf.org/rfc/rfc4034.txt.

[5] D. Atkins and R. Austein. *Threat Analysis of the Domain Name System (DNS)*. RFC 3833 (Informational). Internet Engineering Task Force, Aug. 2004. URL: http://www.ietf.org/rfc/rfc3833.txt.

[6] L. Bassham, W. Polk, and R. Housley. *Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 3279 (Proposed Standard). Internet Engineering Task Force, Apr. 2002. URL: http://www.ietf.org/rfc/rfc3279.txt.

[7] S. M. Bellovin. "Using the Domain Name System for System Break-ins". In: *Proceedings of the Fifth Usenix UNIX Sercurity Symposium* (June 1995). DOI: 10.1.1.178.5616.

[8] R. Braden. *Requirements for Internet Hosts - Application and Support*. RFC 1123 (INTERNET STANDARD). Internet Engineering Task Force, Oct. 1989. URL: http://www.ietf.org/rfc/rfc1123.txt.

[9] R. Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122 (INTERNET STANDARD). Internet Engineering Task Force, Oct. 1989. URL: http://www.ietf.org/rfc/rfc1122.txt.

[10] R. Bush. *Delegation of IP6.ARPA*. RFC 3152 (Best Current Practice). Internet Engineering Task Force, Aug. 2001. URL: http://www.ietf.org/rfc/rfc3152.txt.

[11] D. Conrad. *Indicating Resolver Support of DNSSEC*. RFC 3225 (Proposed Standard). Internet Engineering Task Force, Dec. 2001. URL: http://www.ietf.org/rfc/rfc3225.txt.

[12] D. Cooper et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280 (Proposed Standard). Internet Engineering Task Force, May 2008. URL: http://www.ietf.org/rfc/rfc5280.txt.

[13] A. Costello. *Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)*. RFC 3492 (Proposed Standard). Internet Engineering Task Force, Mar. 2003. URL: http://www.ietf.org/rfc/rfc3492.txt.

[14] J. Damas, M. Graff, and P. Vixie. *Extension Mechanisms for DNS (EDNS(0))*. RFC 6891 (INTERNET STANDARD). Internet Engineering Task Force, Apr. 2013. URL: http://www.ietf.org/rfc/rfc6891.txt.

[15] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). Internet Engineering Task Force, Aug. 2008. URL: http://www.ietf.org/rfc/rfc5246.txt.

[16] D. Eastlake 3rd. *DNS Request and Transaction Signatures (SIG(0)s)*. RFC 2931 (Proposed Standard). Internet Engineering Task Force, Sept. 2000. URL: http://www.ietf.org/rfc/rfc2931.txt.

[17] D. Eastlake 3rd. *Domain Name System (DNS) Case Insensitivity Clarification*. RFC 4343 (Proposed Standard). Internet Engineering Task Force, Jan. 2006. URL: http://www.ietf.org/rfc/rfc4343.txt.

[18] D. Eastlake 3rd. *Domain Name System Security Extensions*. RFC 2535 (Proposed Standard). Internet Engineering Task Force, Mar. 1999. URL: http://www.ietf.org/rfc/rfc2535.txt.

[19] D. Eastlake 3rd. *DSA KEYs and SIGs in the Domain Name System (DNS)*. RFC 2536 (Proposed Standard). Internet Engineering Task Force, Mar. 1999. URL: http://www.ietf.org/rfc/rfc2536.txt.

[20] D. Eastlake 3rd. *RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS)*. RFC 3110 (Proposed Standard). Internet Engineering Task Force, May 2001. URL: http://www.ietf.org/rfc/rfc3110.txt.

[21] D. Eastlake 3rd and C. Kaufman. *Domain Name System Security Extensions*. RFC 2065 (Proposed Standard). Internet Engineering Task Force, Jan. 1997. URL: http://www.ietf.org/rfc/rfc2065.txt.

[22] D. Eastlake 3rd and A. Panitz. *Reserved Top Level DNS Names*. RFC 2606 (Best Current Practice). Internet Engineering Task Force, June 1999. URL: http://www.ietf.org/rfc/rfc2606.txt.

[23] R. Elz and R. Bush. *Clarifications to the DNS Specification*. RFC 2181 (Proposed Standard). Internet Engineering Task Force, July 1997. URL: http://www.ietf.org/rfc/rfc2181.txt.

[24] R. Elz and R. Bush. *Serial Number Arithmetic*. RFC 1982 (Proposed Standard). Internet Engineering Task Force, Aug. 1996. URL: http://www.ietf.org/rfc/rfc1982.txt.

[25] P. Faltstrom and M. Mealling. *The E.164 to Uniform Resource Identifiers (URI) Dynamic Delegation Discovery System (DDDS) Application (ENUM)*. RFC 3761 (Proposed Standard). Internet Engineering Task Force, Apr. 2004. URL: http://www.ietf.org/rfc/rfc3761.txt.

[26] T. Hardie. *Distributing Authoritative Name Servers via Shared Unicast Addresses*. RFC 3258 (Informational). Internet Engineering Task Force, Apr. 2002. URL: http://www.ietf.org/rfc/rfc3258.txt.

[27] K. Harrenstien, M. Stahl, and E. Feinler. *DoD Internet host table specification*. RFC 952. Internet Engineering Task Force, Oct. 1985. URL: http://www.ietf.org/rfc/rfc952.txt.

[28] K. Harrenstien, M. Stahl, and E. Feinler. *Hostname Server*. RFC 953 (Historic). Internet Engineering Task Force, Oct. 1985. URL: http://www.ietf.org/rfc/rfc953.txt.

[29] A. Herzberg and H. Shulman. "Fragmentation Considered Poisonous". In: *Computer Research Repository* (2012). URL: http://arxiv.org/abs/1205.4011.

[30] T. Hlaváček. *IP fragmentation attack on DNS*. RIPE 67. Athens, Oct. 16, 2013. URL: https://ripe67.ripe.net/presentations/240-ipfragattack.pdf.

[31] P. Hoffman and W. Wijngaards. *Elliptic Curve Digital Signature Algorithm (DSA) for DNSSEC*. RFC 6605 (Proposed Standard). Internet Engineering Task Force, Apr. 2012. URL: http://www.ietf.org/rfc/rfc6605.txt.

[32] G. Huston. *Management Guidelines & Operational Requirements for the Address and Routing Parameter Area Domain („arpa")*. RFC 3172 (Best Current Practice). Internet Engineering Task Force, Sept. 2001. URL: http://www.ietf.org/rfc/rfc3172.txt.

[33] "IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) Base Specifications, Issue 7". In: *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)* (2008). DOI: 10.1109/IEEESTD.2008.4694976. URL: http://pubs.opengroup.org/onlinepubs/9699919799/.

[34] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648 (Proposed Standard). Internet Engineering Task Force, Oct. 2006. URL: http://www.ietf.org/rfc/rfc4648.txt.

[35] B. Kaliski. *PKCS #1: RSA Encryption Version 1.5*. RFC 2313 (Informational). Internet Engineering Task Force, Mar. 1998. URL: http://www.ietf.org/rfc/rfc2313.txt.

[36] D. Kaminsky. *Black Ops 2008: It's The End Of The Cache As We Know It*. Black Hat. Japan, 2008. URL: https://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Kaminsky/BlackHat-Japan-08-Kaminsky-DNS08-BlackOps.pdf.

[37] D. Karrenberg. *DNS Root Name Servers Frequently Asked Questions*. Version 2. Feb. 2008. URL: http://www.isoc.org/briefings/020/.

[38] J. Klensin. *IAB Statement on Infrastructure Domain and Subdomains*. May 2000. URL: http://www.iab.org/documents/correspondence-reports-documents/docs2000/iab-statement-on-infrastructure-domain-and-subdomains-may-2000/.

[39] O. Kolkman, W. Mekking, and R. Gieben. *DNSSEC Operational Practices, Version 2*. RFC 6781 (Informational). Internet Engineering Task Force, Dec. 2012. URL: http://www.ietf.org/rfc/rfc6781.txt.

[40] S. Kwan et al. *Generic Security Service Algorithm for Secret Key Transaction Authentication for DNS (GSS-TSIG)*. RFC 3645 (Proposed Standard). Internet Engineering Task Force, Oct. 2003. URL: http://www.ietf.org/rfc/rfc3645.txt.

[41] B. Laurie et al. *DNS Security (DNSSEC) Hashed Authenticated Denial of Existence.* RFC 5155 (Proposed Standard). Internet Engineering Task Force, Mar. 2008. URL: http://www.ietf.org/rfc/rfc5155.txt.

[42] D. Lawrence. *Obsoleting IQUERY.* RFC 3425 (Proposed Standard). Internet Engineering Task Force, Nov. 2002. URL: http://www.ietf.org/rfc/rfc3425.txt.

[43] E. Lewis and A. Hoenes. *DNS Zone Transfer Protocol (AXFR).* RFC 5936 (Proposed Standard). Internet Engineering Task Force, June 2010. URL: http://www.ietf.org/rfc/rfc5936.txt.

[44] C. Macavinta. *AlterNIC takes over InterNIC traffic.* CNET News. July 14, 1997. URL: http://news.cnet.com/AlterNIC-takes-over-InterNIC-traffic/2100-1033_3-201382.html.

[45] P. Mockapetris. *Domain names - concepts and facilities.* RFC 1034 (INTERNET STANDARD). Internet Engineering Task Force, Nov. 1987. URL: http://www.ietf.org/rfc/rfc1034.txt.

[46] P. Mockapetris. *Domain names - implementation and specification.* RFC 1035 (INTERNET STANDARD). Internet Engineering Task Force, Nov. 1987. URL: http://www.ietf.org/rfc/rfc1035.txt.

[47] J. Postel. *Domain Name System Structure and Delegation.* RFC 1591 (Informational). Internet Engineering Task Force, Mar. 1994. URL: http://www.ietf.org/rfc/rfc1591.txt.

[48] J. Postel. *Domain names plan and schedule.* RFC 881. Internet Engineering Task Force, Nov. 1983. URL: http://www.ietf.org/rfc/rfc881.txt.

[49] J. Postel and J. Reynolds. *Domain requirements.* RFC 920. Internet Engineering Task Force, Oct. 1984. URL: http://www.ietf.org/rfc/rfc920.txt.

[50] V. Risk. *ISC concludes BIND 10 development with Release 1.2: Project renamed 'Bundy'.* Apr. 17, 2014. URL: https://www.isc.org/blogs/isc-concludes-bind-10-development-with-release-1-2-project-renamed-bundy/.

[51] RSA Laboratories. *PKCS #11: Cryptographic Token Interface Standard.* 2009. URL: http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm.

[52] R. Shirey. *Internet Security Glossary, Version 2.* RFC 4949 (Informational). Internet Engineering Task Force, Aug. 2007. URL: http://www.ietf.org/rfc/rfc4949.txt.

[53] D. B. Terry et al. *The Berkeley Internet Name Domain Server.* Tech. rep. EECS Department, University of California, Berkeley, May 1984. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/5957.html.

[54] S. Turner. *Asymmetric Key Packages.* RFC 5958 (Proposed Standard). Internet Engineering Task Force, Aug. 2010. URL: http://www.ietf.org/rfc/rfc5958.txt.

[55] P. Vixie. *A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY).* RFC 1996 (Proposed Standard). Internet Engineering Task Force, Aug. 1996. URL: http://www.ietf.org/rfc/rfc1996.txt.

[56] P. Vixie et al. *Dynamic Updates in the Domain Name System (DNS UPDATE).* RFC 2136 (Proposed Standard). Internet Engineering Task Force, Apr. 1997. URL: http://www.ietf.org/rfc/rfc2136.txt.

[57] P. Vixie et al. *Secret Key Transaction Authentication for DNS (TSIG)*. RFC 2845 (Proposed Standard). Internet Engineering Task Force, May 2000. URL: http://www.ietf.org/rfc/rfc2845.txt.

[58] S. Woolf and D. Conrad. *Requirements for a Mechanism Identifying a Name Server Instance*. RFC 4892 (Informational). Internet Engineering Task Force, June 2007. URL: http://www.ietf.org/rfc/rfc4892.txt.

[59] C. Wright. "Understanding Kaminsky's DNS Bug". In: *Linux Journal* (July 25, 2008). URL: http://www.linuxjournal.com/content/understanding-kaminskys-dns-bug.

Appendix A

DNS Messages Format

A.1 DNS Message

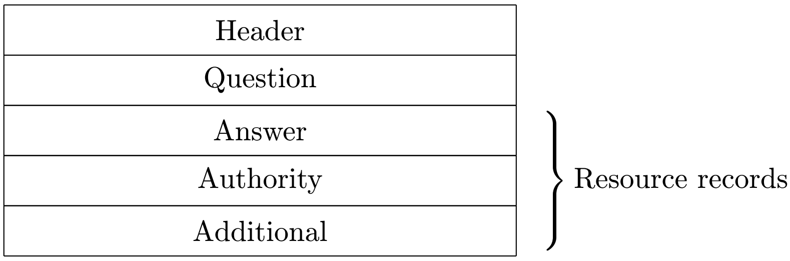

Figure A.1: General structure of a DNS message.

A.1.1 Message Header Format

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0:	ID															
2:	QR	OPCODE				AA	TC	RD	RA	Z	AD	CD	RCODE			
4:	QDCOUNT															
6:	ANCOUNT															
8:	NSCOUNT															
10:	ARCOUNT															

Figure A.2: Wire format of header field in DNS message.

The header has a fixed size of 96 bytes. The purpose of each field:

- The **ID** field holds a transaction identifier set by the client at random. The server copies the identifier into the response message.

- The **QR** is the Query/Response flag. In case the message is a query, the flag is cleared. For answers, the flag is set.

- The **OPCODE** specifies operation code set by client and preserved by server in the reply message.
  - Value **0** means standard query.
  - Value **1** means inverse query. This operation is obsolete [42].
  - Value **2** means status query.
  - Value **4** means notify message [55].
  - Value **5** means update message [56].
- The **AA** (Authoritative Answer) flag indicates that the answer was issued by name server which is authoritative for the domain name in question. The flag is valid only in answers.
- The **TC** (Truncated) flag indicates that the message was truncated due to maximal allowed size of the message. The flag is valid only in answers.
- The **RD** (Recursion Desired) flag indicates that the client desires recursive resolution of the query. The value is preserved by the server in the reply message.
- The **RA** (Recursion Available) flag indicates that the name server is capable of recursive resolution. The flag is valid only in answers.
- The **Z** is reserved for future use and must be zero.
- The **AD** (Authentic Data) flag is ignored in queries. It indicates, that the responding component considers the data in the answer and authority section to be authentic. This flag is was defined in DNSSEC specification.
- The **CD** (Checking Disabled) flag is set in the query and copied into response. It disables DNSSEC checking on the security aware resolver.
- The **RCODE** value is a response code. The content of the field is ignored in questions.
  - Value **0** means No Error.
  - Value **1** means Format Error.
  - Value **2** means Server Failure.
  - Value **3** means Non-existent Domain.
  - Value **4** means Not Implemented.
  - Value **5** means Query Refused.
- The **QDCOUNT** holds the number of entries in Question section.
- The **ANCOUNT** holds the number of RRs in Answer section.
- The **NSCOUNT** holds the number of RRs in Authority section.
- The **ARCOUNT** holds the number of RRs in Additional section.

A.1.2 Question Entry Format

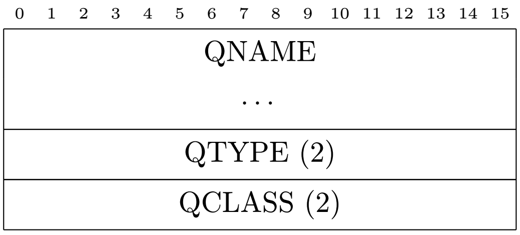

Figure A.3: Wire format of question section entry in DNS message.

The meaning of the fields:

- The **QNAME** specifies the domain name in question.
- The **QTYPE** specifies RR type to be returned.
- The **QCLASS** specifies class (protocol family identifier) of the RR.

A.1.3 Resource Record

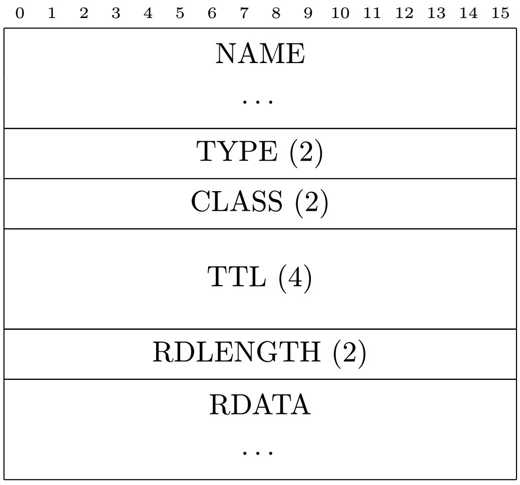

Figure A.4: Wire format of resource record in DNS message.

The meaning of the fields:

- The **NAME** specifies domain name to which the RR belongs.
- The **TYPE** specifies the type of RR.
- The **CLASS** specifies the class of the RR.
- The **TTL** specifies time interval in seconds, how long the RR can be cached and considered valid.
- The **RDLENGTH** specifies length of the RDATA field in bytes.
- The **RDATA** contains the data carried by the RR. The format of the data is determined by the TYPE and CLASS fields.

A.1.4 OPT Resource Record

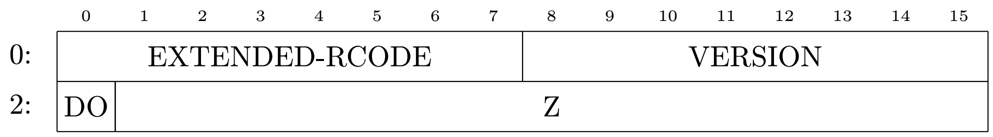

Figure A.5: Format of TTL field of OPT resource record in EDNS.

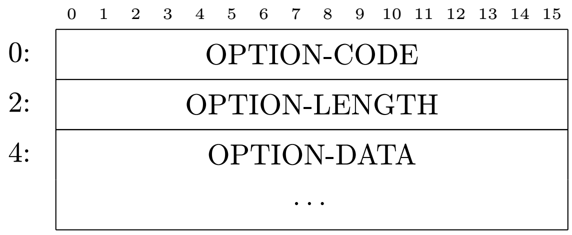

Figure A.6: Wire format of EDNS option, which is used in OPT RDATA.

The content of OPT RR fields:

- The **NAME** value must be the root domain (one byte with value `0x00`).
- The **TYPE** identifies the OPT RR (value 41).
- The **CLASS** has a meaning of **Maximal UDP payload size**.
- The **TTL** was split into multiple fields, described in Figure A.5.
- The **RDLENGTH** specifies length of the RDATA field in bytes.
- The **RDATA** contains a sequence of additional EDNS options. Format of each option is shown in Figure A.6.

The former TTL field contains:

- The **EXTENDED-RCODE** field extends the existing four bits of RCODE in message header for upper eight bits, thus creating 12-bit code. If the value is zero, unextended RCODE is used.
- The **VERSION** field indicates a version of EDNS used.
- The **DO** (DNSSEC OK) flag is allocated for DNSSEC purposes.
- The **Z** field is reserved for future use and must be zero.

The meaning of the option fields:

- The **OPTION-CODE** value identifies the option.
- The **OPTION-LENGTH** value specifies the size of the data assigned to the option.
- The **OPTION-DATA** contains the option data.

A.2 DNSSEC Resource Records

A.2.1 DNSKEY Resource Record

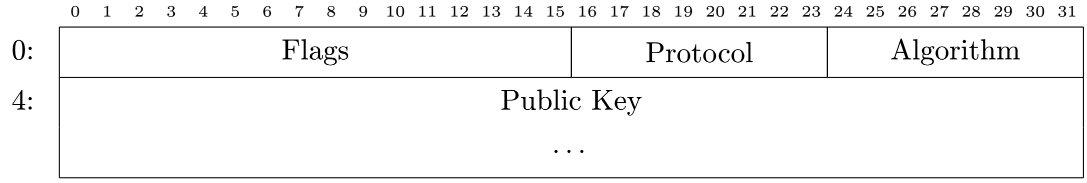

Figure A.7: Wire format of DNSKEY RDATA.

The meaning of the fields:

- The **Flags** field determines the type of the key. Only bits 7 and 15 in the flags are allocated, other bits are reserved and must be zero. Bit 7 is the *Zone Key* flag and if set, the key can be used for verification of RRSIG RRs. Bit 15 is the *Secure Entry Point* flag and it serves just as a hint for resolvers, that the key is most likely used to sign other keys.
- The **Protocol** field must contain value 3, otherwise the key must be ignored.
- The **Algorithm** field contains algorithm identifier. The list of defined algorithms is maintained by IANA.
- The **Public Key** field contains public key material in a format depending on a used algorithm.

A.2.2 RRSIG Resource Record

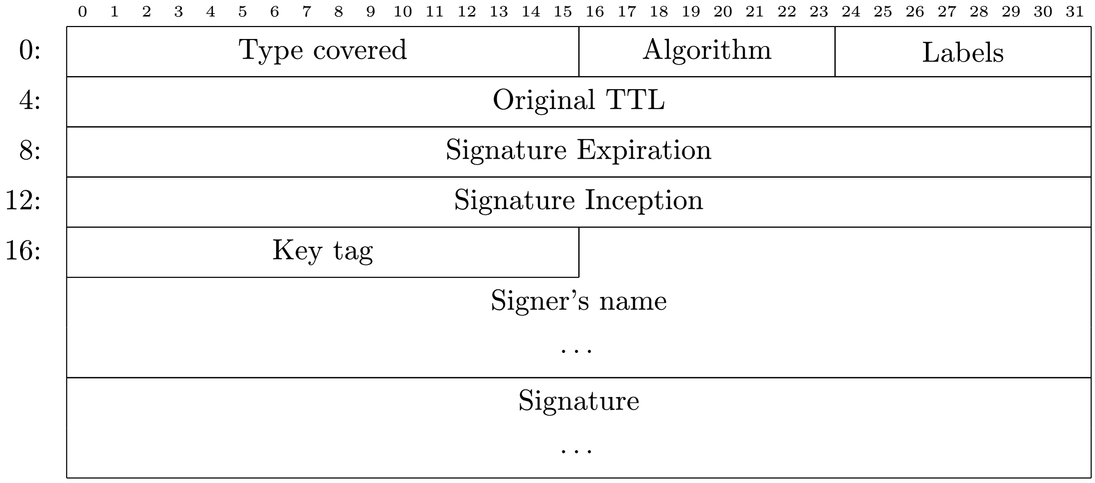

Figure A.8: Wire format of RRSIG RDATA.

The meaning of the fields:

- The **Type Covered** field identifies the type of the RR set, that is covered by the signature.

- The **Algorithm** field identifies the algorithm used to create the signature.
- The **Labels** field contains a number of labels in the owner domain name of the original RR, excluding the root label and wildcard label. The field is used by the resolver to identify RRs synthesised from wildcard.
- The **Original TTL** field holds the TTL value of the covered RR as it appears in the authoritative zone. The original value is required because the caching resolver decrements TTL value of cached RR set, and the validating resolver must be able to reconstruct it's canonical representation.
- The **Signature Expiration** and **Inception** fields reduce the validity period of the signature. Both fields are expressed as a POSIX time (number of seconds since the Epoch [33]). A comparison of these values must be performed using Serial number arithmetic [24], as the time expression can overflow the 32 bit value.
- The **Key Tag** field contains a key tag of used key, which is a simple hash value of DNSKEY RDATA. This field is intended to ease the selection of key for signature validation.
- The **Signer's Name** field contains a domain name of the DNSKEY, which should be used for validation. The name must match with the name of the zone of the covered RR set.
- The **Signature** field contains the digital signature. The format of this field depends on the algorithm in use. The digital signature is calculated from a concatenation of the RRSIG RDATA without the Signature field, and all RRs in the covered RR set, which must be in canonical form.

A.2.3 NSEC Resource Record

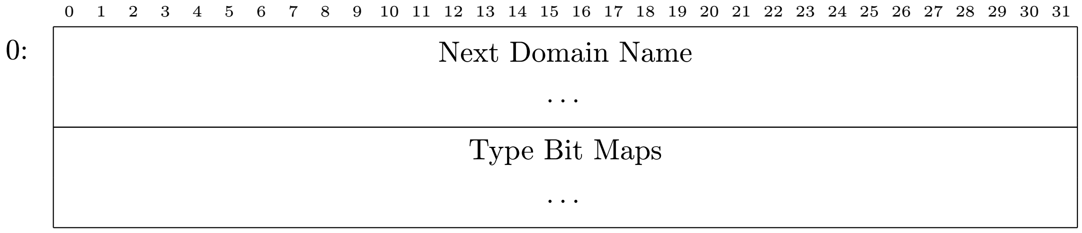

Figure A.9: Wire format of NSEC RDATA.

The meaning of the fields:

- The **Next Domain Name** field contains the owner name of the NSEC RR following the current one in canonical ordering of the zone. If the record is the last one in the ordering, the field contains the name of the zone apex (which is always the first name in the ordering).
- The **Type Bit Maps** field encodes all RR types, which exist in the zone, have the same owner as the NSEC RR, and are authoritative or serve as a delegation point.

  As the RR space is usually very sparse and only a few RR types are used in practice, a special encoding is used. The 16 bit RR type space is split into 256 windows. Each window (8 high order bits) represents a bitmap (8 low order bits), which is 32 bytes long.

Only a windows with at least one active RR, and only a bytes from the beginning of the window to the last used byte of the window are encoded. The window is represented as one byte with the window number, one byte specifying the length of the bitmap, and then individual bytes of the bitmap. The Type Bit Maps field contains a sequence of windows encoded in this form.

A.2.4 DS Resource Record

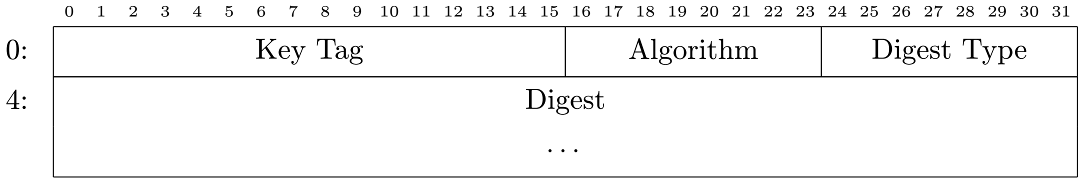

Figure A.10: Wire format of DS RDATA.

The meaning of the fields:

- The **Key Tag** and **Algorithm** fields list the values from exactly named fields of DNSKEY record referred to.
- The **Digest Type** field identifies the cryptographic algorithm used to compute the digest in the next field. It also determines the length of the next field. The only valid value is 1, which means SHA-1 algorithm and 20 bytes long digest.
- The **Digest** field contains a digest computed from a concatenation of the owner name and DNSKEY RDATA.

A.2.5 NSEC3 Resource Record

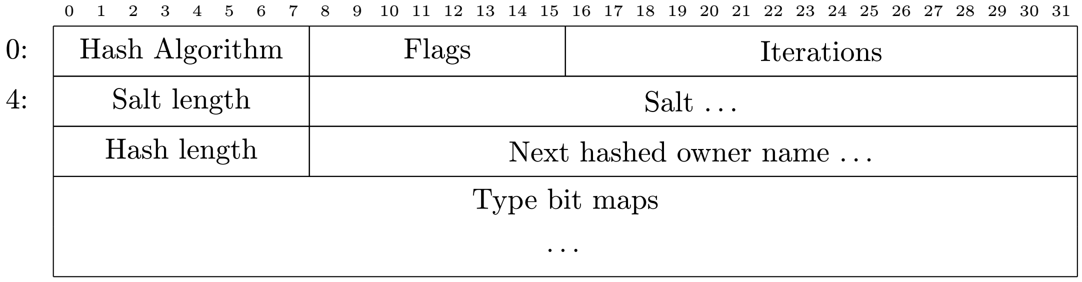

Figure A.11: Wire format of NSEC3 RDATA.

The meaning of the fields:

- The **Hash Algorithm** field identifies the cryptographic algorithm used for hashing. The list of supported algorithms is maintained by IANA.
- The **Flags** field contains flags, which can indicate different processing. Currently only the defined flag is the Opt-Out flag, which is the least significant bit of the field.

If the Opt-Out flag is set, the NSEC3 record covers zero or more insecure delegations. Otherwise it does not cover any insecure delegation.

- The **Iterations** field contains a number of additional times, the hash function is performed.
- The **Salt length** field contains a length of Salt field in bytes.
- The **Salt** field contains a binary data, which are added to the original owner name before hashing.
- The **Hash length** field contains a length of Next Hashed Owner Name field.
- The **Next Hashed Owner Name** field contains the raw hash used to construct a next hashed owner name in NSEC3 chain. If the NSEC3 record is the last record in the chain, the field contains a hash for the first name in the chain.
- The **Type Bit Maps** has the same meaning and uses the same format as in NSEC RRs.

A.2.6 NSEC3PARAM Resource Record

	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0:	Hash Algorithm	Flags	Iterations
4:	Salt length	Salt . . .	

Figure A.12: Wire format of NSEC3PARAM RDATA.

The meaning of the fields is equivalent with the fields used in NSEC3 RR.

Appendix B

Content of the DVD

The attached DVD contains the following directories:

- **sources** — snapshot of Knot DNS source codes with the DNSSEC library;
- **docs** — DNSSEC library interface documentation and samples;
- **tests** — results of the library testing;
- **virt** — virtual image in OVF format with GNU/Debian Linux and preconfigured environment to run the Knot DNS server and to reproduce the tests; and
- **latex** — source code of this thesis in the LaTeX format.