Shaping DNS Security with Curves

A Comparative Security Analysis of DNSSEC and DNSCurve

Master Thesis

Coding Theory and Cryptology
Department of Mathematics and Computer Science
Eindhoven University of Technology

Author:
ing. Harm H.A. van Tilborg

Graduation tutor:
drs. J. Scheerder

Graduation supervisor:
dr. B.M.M. de Weger

Committee:
dr. D.S. Jarnikov
drs. J. Scheerder
dr. B. Škorić
dr. B.M.M. de Weger

August 2010

Contents

Glossary

API Application Programming Interface. A defined interface that makes computer programs (or: applications) able to communicate with each other. Note this communication can also take place over a network, such as the Internet.

byte To avoid ambiguity, a 'byte' stands for an ordered collection of 8 bits in this thesis.

daemon A computer program that is constantly running in the background of a system. Usually waiting for input or requests to be handled and answered.

DNSSEC Domain Name System Security Extensions and an extension to the regular Domain Name System to provide integrity.

domain A domain represents a human readable identifying name on the Internet. If `example.org` is a domain, this means `sub.example.org` and `www.sub.example.org` are considered to be in the same domain. A domain is the superset of a zone.

DoS Denial of Service. Type of attack that influences the availability of a system.

IETF Internet Engineering Task Force. Open standards organization of the Internet. The IETF facilitates the process of developing new RFCs, from drafting and reviewing, until testing and publishing.

IP Internet Protocol. This is the layer between the Internet's network layer and link layer. It also facilitates the addressing of networks and hosts around the Internet. There are two branches of IP: IPv4 and IPv6. The biggest difference between the two is the addressing space. IPv4 offers space for 2^{32} hosts, while IPv6 can address 2^{128} hosts.

ISP Internet Service Provider. Often a commercial company that facilitates connections towards the Internet for private persons or organizations.

KSK Key Signing Key. Key that is used to authenticate other (usually zone signing) keys. Generally located in the same zone as the keys it authenticates. See also ZSKs.

LAN Local Area Network. A network that connects hosts that are physically relatively close to each other, for example in one's home, an office, or a group of buildings. Opposite of WAN.

MAC Message Authentication Code. Also known as keyed cryptographic hash function. Hash function that together with a secret key can generate a fixed size string that provides data integrity and authenticity of an arbitrary length piece of data.

NaCl Networking and Cryptographic library, pronounced as 'salt'. Library that implements all DNSCurve used cryptographic primitives used by DNSCurve.

nonce Number used ONCE. An arbitrary number of cryptographically random bytes, used to guarantee freshness in (security) protocols, preventing replay attacks.

POSIX Portable Operating System Interface for Unix. A family of standards developed by the IEEE to accommodate a general application programming interface between user space and an operating system kernel. Originally focused on the Unix operating system, the principles can however also apply to other operating systems.

qps Queries per Second. The number of queries that a service can handle per second.

RFC Request for Comments. Formal method of publishing Internet protocol standards, facilitated by the IETF. Besides referring to the process itself, the term 'RFC' also is a document that describes such Internet protocol.

RIPE Réseaux IP Européens. Part of RIPE is the RIPE NCC, which is a Regional Internet Registry (RIR). This organization is responsible for the allocation of IP addresses in Europe and the Middle East.

RR Resource Record. A resource record is part of a DNS response, and usually grouped by its resource record type.

RRset Resource Record set. A resource record set is a set of resource records (RR) that is grouped by its response name, type, class, and sometimes – depending on the situation – by its TTL.

TLD Top Level Domain. These Top Level Domains are located directly under the DNS root. A subset of these TLDs are called the generic top level domains (gTLDs). Examples of these generic domains are the `.com` and `.gov` domains. Besides also country code top level domains (ccTLDs) exist, examples hereof are `.nl`, `.uk`, and `.au`.

TTL Time To Live. A timing constraint on some piece of data, usually given in the number of seconds this piece of data can be considered valid.

VM Virtual Machine. A Virtual Machine is used as a middle-ware between the operating system and a certain program. Virtual Machine based programming languages are for example Java and Python. VMs take over memory management of a program, and can garbage collect unused memory. Because of these checks, it creates a safe environment, although it also decreases performance.

WAN Wide Area Network. A network that connects hosts that are physically broadly divided. Opposite of LAN.

zone A zone is the part of a domain that is served by an authoritative name server. Take for example the domain `example.org`. The subdomain `sub.example.org` also exists. However the latter one is served by another name server. This means the `example.org` zone does not include `sub.example.org`. A zone is always a subset of the corresponding domain.

ZSK Zone Signing Key. Key that is used to authenticate zone data. See also KSKs.

Acknowledgments

“Standing on the Shoulder of Giants”

I always liked this metaphor, although I do not know whether Oasis had something to do with it. Nevertheless, throughout my short life so far I have applied it not solely to academical matters. Just like I will not do this, this time.

Without the help by many of these giants, I would not have been so proud of my master project – and this master thesis in particular – as I am now. Therefore I would like to thank Daniel Bernstein, Matthew Dempsky, and Adam Langley, for their support, code examples, and interesting discussions on many DNSCurve related matters. Without their help, many things would not have been discussed in so much detail in this thesis. Next, I would like to thank Marc Lehmann, for answering many questions in relation with his great libev library. Furthermore, the DNSSEC chapter would not have been so covering without the help of Matthijs Mekking, thanks for that.

A good project is nothing without someone having a good overview from above, together with a sharp eye for missing details. Therefore, I would like to thank my graduation supervisor Benne de Weger.

Furthermore, I would like to thank all my colleagues at ON2IT B.V. Firstly, because of giving me the opportunity to use their expertise and facilities during my master project. And secondly, to thank them for the nice atmosphere during the days I was there.

In particular I would like to thank Jeroen Scheerder for being my graduation tutor. His support positively influenced the project as well as the thesis, increasing the quality of both.

Finally, I would like to thank everyone interested in my project along the way: classmates, colleagues, family, and friends. And last, but certainly not least, my thanks go to both my parents for their support in every imaginable way for the last 23 years: Pap & Mam bedankt!

Harm van Tilborg, Sint Anthonis-Waardenburg, May–August 2010

List of Figures

List of Tables

Chapter 1

Introduction

"It's long been said that the revolutions in communications and information technology have given birth to a virtual world. But make no mistake: This world – cyberspace – is a world that we depend on every single day. It's our hardware and our software, our desktops and laptops and cell phones and Blackberries that have become woven into every aspect of our lives.

It's the broadband networks beneath us and the wireless signals around us, the local networks in our schools and hospitals and businesses, and the massive grids that power our nation. It's the classified military and intelligence networks that keep us safe, and the World Wide Web that has made us more interconnected than at any time in human history.

So cyberspace is real. And so are the risks that come with it."

President Barack Obama on Cybersecurity – May 29, 2009 [49]

What more can be added to this introduction, that perfectly describes what aspects in our lives have become reliant on information technology, and communications in particular. Besides the huge benefits it brought us, there are also important downsides to be recognized. Connecting networks gives opportunity to persons abusing the available infrastructure in a way that could negatively influence any process trusting this infrastructure. Organized crime syndicates, competing businesses, ordinary criminals, enemy considered nations, or your next door neighbor, the list of possible attackers is endless.

It is obvious all these threats should be mitigated to an absolute minimum. With the Internet being the largest network that we know – that is open to everyone, from school children to ministers, and from parents to academics – giving security related guarantees would be of great value.

This master thesis will focus on one of the Internet's most important protocols, the so called Domain Name System (abbreviated to 'the DNS'). The

Domain Name System is responsible for translating a domain name to an Internet address. This means that without the DNS, the Internet will not work as we expect it to work. Because the DNS is so crucial in the functioning of the Internet, it should be protected to ensure its availability and integrity. It will become clear it fails to do this at some points, therefore additional protocols have been developed. This thesis will discuss two different and independent security extensions of the DNS.

1.1 Previous Work

Many has been written about the DNS in general, ranging from specification [44] [45] and documentation [3], to new functionality proposals [9]. At the beginning of the 1990s, more focus was given on the protocol's security [15] [16]. Steven Bellovin wrote a paper called "Using the Domain Name System for System Break-Ins" [16] in 1990. It was however withheld from publication because – in the writer's opinion – no fixes to the discussed flaws were available until 1995.

This caused to start of development of a new, enhanced DNS protocol that is able to stop all the risks the regular DNS suffered. This protocol came to be known as DNSSEC [12] [13] [14], and tries to supply a secure extension to the standard DNS. Also for DNSSEC there is plenty of material available [5] [8] [35] [47].

DNSCurve is an alternative to DNSSEC – although both are not mutually exclusive. Because the DNSCurve protocol is relatively new, compared to the DNS as well as DNSSEC, not much material is available that discusses the protocol yet [6] [30].

1.2 Problem Statement

This master thesis is constructed among the following research question:

> *Given the problems the regular DNS suffers and the proposed solutions to these problems DNSSEC and DNSCurve, do these solutions actually succeed in solving the problems (1), do they themselves introduce any new problems (2), and are both solutions feasible to be used in current and future practice (3)?*

To come up with answers for each of these subquestions, several objectives have been specified, that will help in answering each of the subquestions.

Because of the existence of two protocols that try to solve the same problems of an earlier protocol, it is good to recognize the differences between both protocols. To be able to do this, a sound understanding of the earlier protocol (the DNS) is needed. Therefore, this thesis will start with a chapter introducing and explaining the Domain Name System. It will also discuss the problems it suffers that have been found over the years.

Next, DNSSEC will be introduced and the problems of DNS will be discussed again, only now in relation to DNSSEC. The problems that DNSSEC might bring will also be discussed. The same is done for DNSCurve. Although, besides this, also a general comparison and a specific security comparison will be performed.

Furthermore, the practical situation will be explained that discusses the characteristics of DNSCurve in practice. Finally, the implementation of a DNSCurve piece of software will be discussed and analyzed afterwards.

This brings us to stating the objectives of this master thesis:

1. Introduce and explain the regular Domain Name System, plus the problems it suffers;

2. Introduce and explain DNSSEC, including the problems it solves and introduces;

3. Introduce and explain DNSCurve, including the problems it solves and introduces;

4. Give a comparative general analysis and specific security analysis of DNSSEC and DNSCurve;

5. Explain the characteristics of using DNSCurve in practice (i.e. deployment, administration, and performance);

6. Design, implement, test, and analyze a DNSCurve implementation.

1.3 Results

Chapter 2 will introduce and discuss the Domain Name System (DNS). In chapter 3 a security related extension to the DNS will be introduced and explained. After this introduction, DNSSEC's security will be related towards that of the regular DNS. Next, chapter 4 will introduce and discuss DNSCurve – also a security extension to the regular DNS. The security of DNSCurve will first be related to that of the regular DNS, while afterwards DNSCurve will be compared with DNSSEC. Chapter 5 will discuss the practical side of DNSCurve. It describes the design, implementation, and testing process of the development of a forwarding DNSCurve capable name server. Afterwards, the implementation is analyzed in its performance by several benchmarks. Finally, chapter 6 will end this thesis with a conclusion.

Chapter 2

Domain Name System (DNS)

This chapter is about the Domain Name System, what it did and does, and eventually what it does not do well. This means flaws in either design or – maybe even more important – security.

2.1 Introduction

In this section the Domain Name System is introduced. This is done by comparing the DNS with the already known telephone numbering system.

2.1.1 Telephone Numbering System

When one wants to phone someone, this person's telephone number is needed. Most people know some of the phone numbers they call often by heart. However, if a person has to call someone he does not call regularly, or maybe even someone he has never actually met, getting the right telephone number is harder. Public phone books can make this process much easier. If the name of the person one wants call is known, preferably together with this person's address, one is able to find this person's phone number in a telephone book.

Now let's relate this story to the Internet. As the name suggests the Internet connects networks to each other. Networks are distinguished by their network address, while hosts living in these networks are identified by their host address. To make communication between networks (and hosts) possible, several communication protocol standards have been proposed and used. Although, the most widely used communication and addressing protocol nowadays is the Internet Protocol (IP) [54].

The IP was proposed at the end of 1981, and later referred to as IPv4. This initial version of the IP is able to address 2^{32} hosts. (Its successor

IPv6 [29] will be able to address 2^{128} hosts.) An address in the IP is represented by a 32 bits number. To represent this 32 bits number, it was chosen to split the 32 bits number into four 8 bits numbers. These kind of addresses are called IP addresses. Examples of such addresses are `127.3.11.86`, and `192.168.14.12`.

If information has to be shared over the Internet, one needs the address of the host supplying this information. So to visit the popular Google search engine for example, one would have to direct its request to `74.125.79.99`.

Now the same problems arises that existed with telephone numbers. It is unfeasible for human beings to know by heart what information is served by what address. This is where the Domain Name System comes into play. The DNS can be compared with the public telephone book. It simply relates human readable names to IP addresses. However, responses are not limited to addresses only, also other information can be returned by the DNS.

Let's go a bit more into detail. The worldwide used telephone 'addressing system' is based upon country codes. For example to make a phone call from another country towards the United States or Canada, first `+1` has to be dialed. Similarly, to call someone in The Netherlands from another country, one needs to prefix that number with `+31`. Next, beyond the – sometimes implicit – country code usually a regional dial code can be found. For example, to call someone in The Netherlands in the region of Eindhoven, the number should be prefixed with `040`[1]. Finally the 'regular' number is attached to the telephone number. Concatenating all separate numbers results in a phone number that can be called.

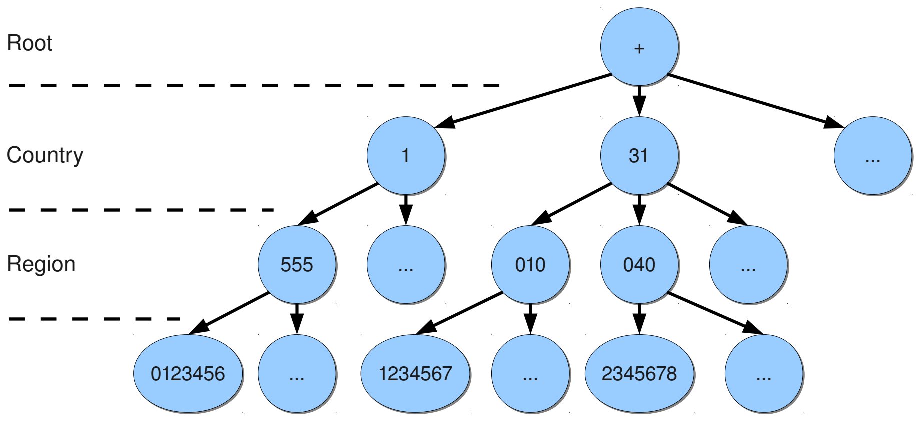

Figure 2.1: *Hierarchical structure of the telephony system*

If this addressing system is considered more abstract, a tree structure can be seen in it. Figure 2.1 illustrates this structure. Notice the tree structure is rooted by `+`. One traversal down the root, all country codes can be found.

[1]To be totally correct when the number is prefixed with `+31` the first zero is omitted, and the start of the number looks like `+31 40`.

Next, there are the region codes and finally the regular numbers. Putting all traversals originating from the root down to the leaves together, the number that someone wants to call can be obtained, for example: `+ 31 040 234578`.

2.1.2 Hierarchical Distributed Database

Now look at this from the perspective of the telephone carrier. It would not be viable for a telephone carrier to know every telephone number around the globe and its respective routing. Therefore a telephone carrier that is handling traffic at the root of the telephone infrastructure, only needs to know what country codes there exist, and what national telephone carrier will handle that country's specific telephone traffic. This list is relatively small and easy to maintain and store. Whenever it knows what country the call has to be directed to, it will forward the request towards that country's national carrier.

When one wants to call the number from the end of the previous section, the Dutch (`+31`) telephone carrier would further handle our telephone traffic, after being delegated there by the root telephone carrier. This national carrier has its own regional list of (perhaps local) telephone traffic handlers. But again, this list is maintainable. Finally, such a local carrier is aware of all its clients and their respective regular phone numbers. They will route the phone call towards the right client.

So participants in this telephone network know only information about its parent and direct children, nothing more. For example the regional carrier in the United States or Canada (i.e. `+ 1 555`) does not know anything about the regional split of Rotterdam (`+ 31 010`) and Eindhoven (`+ 31 040`) in The Netherlands. It is only aware there is a `0123456` number in the `555` region, and its parent is the one handling all `+ 1` telephone traffic. This phenomenon is called a hierarchical database, since a node in the tree only has knowledge about nodes one place up or down in the hierarchy. Because the information where the telephone call has to go to, is not centrally located somewhere, this database is also decentralized. Therefore this database is called a hierarchical distributed database.

Let's go back to the DNS. As mentioned, the DNS was designed to accommodate the need to translate a human understandable name (or: string) to an IP address. Knowing there could be around 2^{32} hosts in the entire IPv4 network, it would be unfeasible for all hosts on the network to keep up with constantly maintaining and transferring this database. Therefore the DNS came up with a referencing scheme. This referencing is done by using parts of the human readable string and is also rooted by a part of the string. Nevertheless, even keeping track of all references would be unfeasible. Therefore the DNS also uses a hierarchical distributed database. Referencing in the DNS means that whenever a host is asked to translate a string, but it does not have the actual answer, it could give an answer like: 'I do not know the address of this string, but you might ask that host over there'. And to accommodate the storage problem: the only one reference that is kept by a host is a reference towards the root. Knowing the root is able to help in translating all existing strings.

Notice this section is a brief and simple introduction to what the DNS is. More detail will be given later in this chapter.

2.2 History

When talking about the history of the Domain Name System, the history of the Internet itself is implicitly discussed. A good overview of what organizations and persons were involved during the rise of both the Internet and the DNS is explained in [3]. This section is based on that document.

Precursor of the Internet as it is known today, was the so called ARPANET. This network was an outcome of research funded by the U.S. Government to develop packet-switching technology and communications networks. The network was established by the Department of Defense's Advanced Research Projects Agency (DARPA) in the 1960s. Later on the ARPANET was linked to other networks created by other government agencies, universities and research facilities. During the 1970s, DARPA also funded the development of a 'network of networks', this being the first glimpse of what is now known as the Internet. In that time also the earlier mentioned Internet Protocols were developed.

Parts of the ARPANET development work were contracted to the University of California in Los Angeles (UCLA). Dr. Jon Postel – at that time a graduate student at UCLA – undertook the maintenance of a list of hostnames and addresses. These lists were made available for the network community by SRI International, under contract to DARPA. This is what can be called the first startup of a naming system. Such a list is still present at modern operating systems in the form of a file and is generally referred to as the `hosts` file (or: `hosts.txt`). This file consists of lines containing the hostname followed by an address. Due to the fact not many hosts were connected, this list was maintainable by a single person.

When Dr. Postel moved from UCLA to the Information Sciences Institute at the University of Southern California (USC), he continued to maintain the list of assigned hostnames, still under contract with DARPA. As the list grew, DARPA permitted Dr. Postel to delegate additional administrative aspects of the list maintenance to SRI, although being under continual technical oversight. Eventually all these administrative functions became collectively known as the Internet Assigned Numbers Authority (IANA). (At present IANA is still active and responsible for the assignment of IP addresses and management of the DNS root zone – which will be discussed later on.)

Until the 1980s the Internet was managed by DARPA, and primarily used for research purposes. Nonetheless, maintenance of the name list became harder. The Domain Name System was developed to make this process better and easier. Dr. Postel and SRI participated in DARPA's development and establishment of the technology and practices used by the DNS. By 1990, ARPANET was completely phased out, including the file based naming system.

The DNS as it is used nowadays was proposed by Dr. Paul Mockapetris at the end of 1987, at that moment also working at the Information Sciences Institute. Its exact specification is discussed in two RFCs: RFC 1034 [44] and RFC 1035 [45]. The first one discusses the DNS protocol in general, while the latter one specifies the protocol in technical detail. Later on many more RFCs followed, correcting or clarifying these first two RFCs and sometimes to bring up new features and/or extensions to the regular DNS.

Before both final RFCs came out, some other RFCs regarding addressing – later on obsoleted by RFC 1034 and RFC 1035 – introduced the hierarchical naming system that is known nowadays. Instead of the country codes as the ones known from the telephone numbering system, the DNS initially made a different choice regarding the top level numbering. They came up with eight Top Level Domains (TLDs), which are later referred to as the generic TLDs. These generic TLDs include the still popular `.com`, `.net`, and `.org` domains. Besides these, there are other generic TLDs that could only be used for U.S. governmental purposes, such as the `.gov` and `.mil` domains.

In the 1990s the Internet started to evolve from a mainly academic nature to a more global purpose. Not only universities and research labs got a connection, but Internet became available to the public. Nowadays there is still ongoing debate about the influence of the U.S. government regarding the Internet, this is beyond the scope of this thesis, [3] does however give a good overview of this subject.

2.3 Specification

Now that is known what the DNS is in general, and how it evolved, it is time to see how it exactly works. This section starts by explaining how domain names are built up, next it will discuss the separation of tasks inside the DNS. Afterwards a DNS traversal will be explained, followed by a general used topology and its corresponding terminology. This section is ended by discussing the exact technical details of the protocol.

2.3.1 Domain Names

In the introduction of this chapter, the DNS was compared with the world-wide used telephone numbering system. A telephone number is split up into several parts, such as the root, the country code, followed by the region code and finally the number itself. Combining all these parts yields the telephone number again.

The biggest difference between both systems is that the 'address' of a telephone number, is directly the way it is used by the caller. The person's name does not hint at the person's telephone number, while in the DNS the domain name refers to an IP address, but no one addresses its webpage request by an IP address.

As already mentioned, it is infeasible for a host to keep the entire translation map from domain names to IP addresses. Therefore the DNS also uses a hierarchical distributed database. This implicitly requires the usage of references, i.e. knowledge of what data can be found at what host.

This is where the structure of a domain name comes into play. Take for example the domain name `foo.example.org`[2] that can be split up, to be seen as a tree. Figure 2.2 shows the tree representation of this domain name, together with other domain names. In this figure the edges indicate the reference a certain node has to another node. For example, the '.' node has knowledge where to find more information about the `.org` node, et cetera.

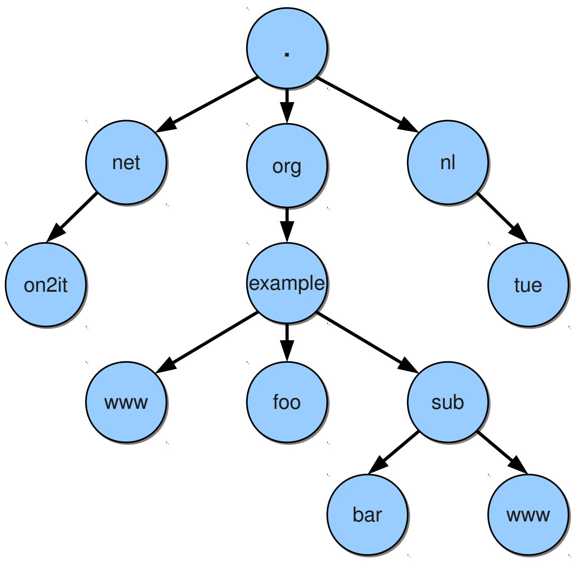

Figure 2.2: *Tree representation of Fully Qualified Domain Names (FQDNs)*

[2]To be precise according to the DNS specification the fully qualified domain name would be `foo.example.org.` – notice the suffixing dot. Please remark that throughout this document both ways of representing a domain name are used.

The '.' node of Figure 2.2 is the most important part of the DNS. This node is referred to as the root of the DNS. As could already be seen in the example, this root node is able to refer to domains under the `.org` node. However, there are many more domains the root node can refer to. In the current Internet environment the root can for example also refer to nodes that are aware of domain names under the `.net`, `.museum`, `.gov`, or `.nl` node. Notice however, the root node is not aware where to find `foo.example.org` or even `example.org`, it only knows that the `.org` node can help one further. This implies the root node does not have a transitive closure of the entire tree (which indeed would be the same as having the entire translation map).

2.3.2 Separation of Duties

Hosts that use the DNS, or are part of the DNS, can be distinguished into two groups. In the first group hosts that house small parts of the translation map can be found. What these small parts are, is defined by the host's authority, i.e. hosts are not allowed to serve information for any domain. How this authority is determined, will be explained later. Hosts in this group will be referred to as the 'authoritative name servers' (note that also the term 'name servers' is used in this document).

Hosts in the other group are interested in the information stored at these name servers. Hosts in this group are called the 'clients'. Later on, in the topology part of this section, this division will be more fine-grained, but this will do for now.

It is clear that clients will ask questions about a translation to the name servers. These questions are called 'queries', while answers to these questions are called 'responses'.

Until now things have been very abstract about what kind of questions a client can make, this knowledge is however essential in the understanding of the DNS. Every query that is sent to a name server, is accompanied by two category identifiers. The first and broadest category identifier is called the query class. The class distinction has mainly a historic meaning, since all separate networks that existed, had their own class. However, most queries take place in the Internet class (shortened to `IN`) nowadays. The next category identifier is the query type field, that indicates what kind of answer the client wants to retrieve. In RFC 1035 [45] sixteen standard types have been defined, in later RFCs many more have been added, although the most important ones that are being used inside this thesis are defined in RFC 1035. Mainly A-, AAAA-, and NS-type queries and responses are used, which stand for the retrieval of an IPv4-address, an IPv6-address, and an authoritative name server domain name respectively. Responses are bundled by their type, and referred to as A-type records, when one or more responses to an A-type query are being discussed.

Earlier it was mentioned how name servers can respond with a reply like: 'I do not know where to find *X*, however name server *Y* might know'. These responses have the type NS, and semantically mean: this is the authoritative name server for this name. NS-type records can therefore be seen as referrers to other name servers. A-type records on the other hand translate a domain name to a 32-bits IPv4 address, while an AAAA-type (pronounce 'quad-A') record does the same for a 128-bits IPv6-address.

Now that a distinction has been made between clients and servers, and it is known what generic query and response types there are, all knowledge is available to understand a DNS traversal.

2.3.3 Traversal

Assume the client wants to resolve the IPv4 address of `foo.example.org`. It should start by querying the root node of the DNS tree and recursing down until it finds a satisfying answer or a negative answer of course. Such recursion down the tree, is known as a DNS traversal (or: recursion). This recursion is graphically portrayed in Figure 2.3[3].

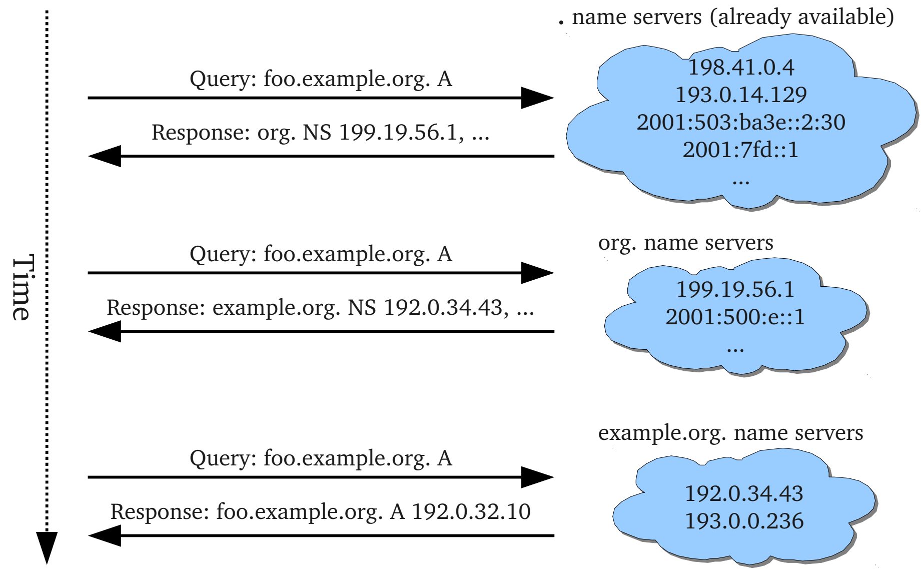
. name servers (already available)
198.41.0.4
193.0.14.129
2001:503:ba3e::2:30
2001:7fd::1
...
Query: foo.example.org. A
Response: org. NS 199.19.56.1, ...
org. name servers
199.19.56.1
2001:500:e::1
...
Query: foo.example.org. A
Response: example.org. NS 192.0.34.43, ...
example.org. name servers
192.0.34.43
193.0.0.236
Query: foo.example.org. A
Response: foo.example.org. A 192.0.32.10
Time

Figure 2.3: *Graphical representation of a DNS traversal*

The root node in the DNS is represented as a set of name servers simply known as the 'root name servers' (or just 'root servers'). Notice that clients should know the IP addresses of these servers by heart, a lookup by domain

[3]Note that some details are left out for simplification (i.e. NS records return IPv4 addresses, and NS records are 'regular' responses, instead of holding up in a response's authority section).

name would fail since there is no starting point. This is also known as the chicken and egg problem.

A client will therefore start to ask an A-type query for `foo.example.org` to one randomly-picked root server. The root server will not reply with A-type records, as the name server has no knowledge of `foo.example.org` – not even of `example.org`. It will therefore reply a set of NS-type records, meaning that servers mentioned in these NS records know more. However, these NS-type records do not include `foo.example.org` as domain name, but only `.org`, as that is the only knowledge the root servers have about the queried domain name.

Since `.org` also appears in the domain name that is queried, a new query to one of the NS-type name servers – that were received from the previous query – is about to be made, to found out if one of these knows more about `foo.example.org`. In Figure 2.3 this is done by sending the query to `199.19.56.1`, which is one of the name servers of the `.org` generic TLD. Besides the gTLDs also country specific top level domains exist. Examples hereof are the `.nl`, `.uk`, `.cn`, and `.au` TLDs, which are known as ccTLDs (country code top level domains). There exist however many more [2].

Getting back at the query to the `.org` top level name server, the answer will again not be an A-type response. It responds with a set of NS-type records, that are authoritative for the `example.org` domain name.

So a new iteration is made, and the `foo.example.org` query is send to one of the name servers, received from the previous query as NS-type records. In the figure this is done by sending the query to `192.0.34.43`. Finally an A-type response is received, stating that `foo.example.org` lives at `192.0.32.10`.

Notice that during this traversal everything went okay, i.e. every name server responded, and every query was positively answered. For each problem that might occur scenarios have been specified how to deal with this. Notice that NS-type records do not consist of addresses, but of domain names themselves. So a lookup for each of these retrieved authoritative name servers should be done in the meanwhile. For the simplicity of this text however, all these details were left out.

2.3.4 Topology

Now that there is a global oversight of a DNS traversal, some observations can be made.

Something that has not been mentioned during the explanation of both the domain name structure, as well as the DNS traversal, is that there is also a timing constraint attached to the retrieved records. This constraint is known as the TTL (time-to-live), and states that a client is allowed to cache (or: temporarily save) the record for this TTL-amount of time. This is performance-wise an enormous improvement. In the time a certain response of a query is cached, no new queries with the same question have to be send towards a name server. Nowadays a TTL is around 12 hours, although earlier – when bandwidth was expensive and slow – this happened to be more than a day.

This caching gets more effective when more clients make use of it. In this case a name server will be able to cache more information, but even more important: it will be able to cache the same start pattern of a DNS recursion. Because every recursion starts at one of the root name servers, followed by one of the top level domain name servers, both remain the same for a long period. This would imply not querying the root and higher level domain name servers again, until the TTL of one of the top level root servers would expire. This benefit quickly reduces the time in which a query can be adequately handled, since records can be pulled directly from the local cache, instead of being queried over the network.

So combining a cache brings great benefits, this is the reason why Internet service providers (ISPs) all have their own 'caching name servers'. This gives a more fine-grained description of both name servers, and clients. On the one hand there are 'authoritative name servers', servers that only serve authoritative DNS data. While on the other hand there are 'caching name servers' (also later referred to as 'resolving name servers' or 'recursive name servers'), that cache DNS data and, if a certain query is not available in cache, they will generally retrieve this data recursively from authoritative name servers. Remark that a caching name server also is a client: it receives and caches data it received from authoritative name servers. However, it is also a name server because clients ask questions to this caching name server and the server responds to them.

Customers of ISPs typically use the ISP's caching name servers as their name server. This has several advantages for both the ISP as well as for the customer. The customer does not have to run a cache on its own now, plus the ISP has faster connections to the Internet, and thus to other name servers. It therefore performs better for the customer. And it saves bandwidth for the ISP, as the same query will now be handled either by the cache or only send once over the Internet and cached afterwards. This same story goes up for organizations. Most organizations use their own caching name server in their network, benefiting the same advantages ISPs do.

Clients that originally initiated the query – i.e. systems of the customer – send their DNS queries to such a caching name server, that will always answer the question; even when a negative response is found. These clients are called 'stub resolvers'. As they do not have complicated algorithms to resolve the answer to a question, they simply pass the query to a caching name server. The response received from this recursive name server is then passed to the program on the customer's system that needed the translation, such as a mail daemon, or web browser for example. Usually these stub resolvers are implemented inside the system's operating system, and can be used by programs by making specific system calls.

Now that the different parts of the DNS have been discussed, it is time to see how this looks when they are all combined. Figure 2.4 illustrates how the various parts fit together.

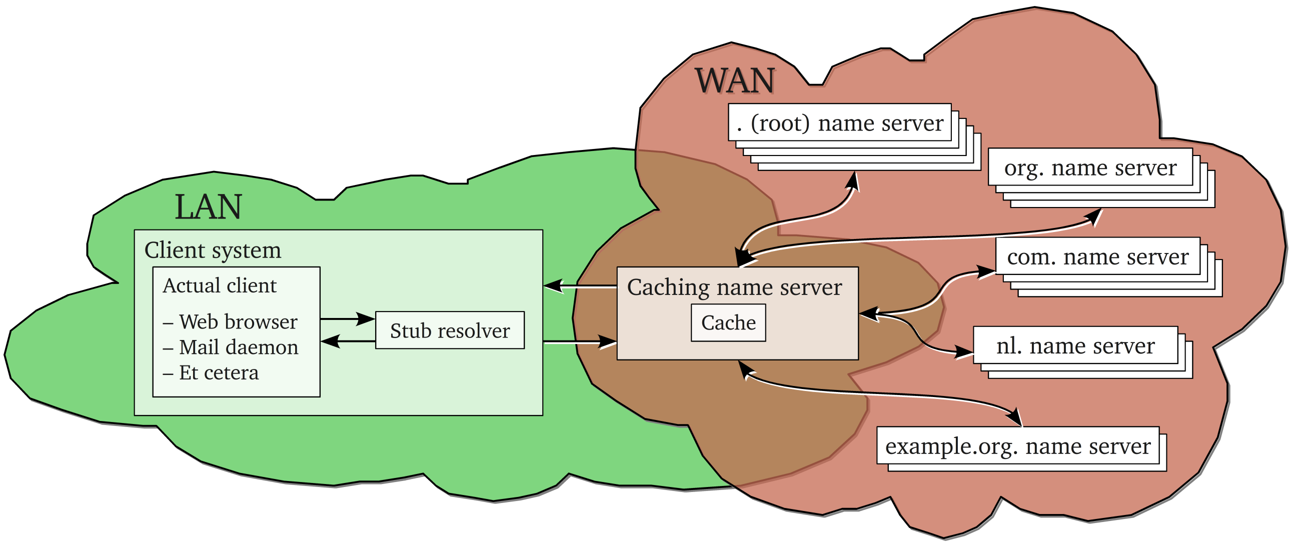

Figure 2.4: *Graphical representation of the DNS topology*

The green- and red cloud are the Local Area Network (LAN) and Wide Area Network (WAN) respectively. There is overlap between both, as explained earlier: sometimes the resolving (or: caching) name server is located on the LAN, for example in an organization. However, in some cases this recursive name server is situated on the WAN, at an ISP for example.

Now look at how all the components are connected. Let's start with looking at the 'client system' block. Inside the client system two different parts can be distinguished: the first one being the application that needs to do a DNS lookup, the second one the so called stub resolver. The connection between both elements is done by the system's operating system. Remark the stub resolver and even some applications also keep (small) local caches of DNS data, although the TTL for these data is reasonably low.

Next, consider the connection between the client's system and the caching name server. Depending on what caching name server is used, this connection can reside on the LAN, but if for example the ISP's resolving name server is used, the connection will be routed over the WAN. Note that these setups can be mixed. An organization could for example use its own in-network (so on its LAN) caching name server, that communicates with the ISP's recursive name server. In this way two separate caches are in place.

Now look at the connections between the authoritative name servers (such as the root name servers, the top level domain name servers, and the `example.org` name server) and the caching name server. Every connection between both servers is made on the WAN. The more a caching name server will cache, the less use is made of the authoritative name servers on the WAN. However, if a query and its corresponding (positive or negative) result is not in the cache, a DNS traversal will be started. Still, the starting path of this DNS traversal will have similarities with previous queries, making the cache useful anyway.

Also notice the number of authoritative name servers there are. Every name server is supported by one or more 'mirror' servers. In this way a fail-over is created. If one of the name servers is down or unreachable, the other ones would still be able to serve the naming information. More about this availability feature will be explained in the security section of this chapter.

2.3.5 Technical Details

At this point an overview of the DNS has been given. However, no technical details were involved. This subsection will change that, it will explain the technical details of the DNS. This is done by explaining the protocol format that is sent over the wire, and discussing all the features and caveats any of the design choices bring. Notice this section is almost entirely based on the two initial DNS RFCs [44] [45].

Header Format

To explain the inner workings of the DNS protocol, it is good to start by explaining the DNS header format. Every DNS packet is build up the way this format prescribes. This means a DNS packet can be anything: a question or a response of any class or type. So to know what the DNS packet exactly is, will be specified by creating the packet's header.

Figure 2.5 illustrates the DNS packet header format. Notice this figure is horizontally aligned by a 2-byte origin. This means one row can store 16-bits of data. So for example the `ID` field can contain a value in the range of $[0, 2^{16} - 1]$, while the `OPCODE` field has four data bits available.

In the paragraphs below, each of the different items corresponding with the DNS packet header will be discussed. When items are not discussed, this means these items are not important for this thesis and will therefore not be treated.

Identifier

The first two bytes of the DNS header identify the packet to make it distinguishable from other packets. This means that both a client and a server can identify DNS packets by using this 16 bits identifier. When a packet is a response (see next paragraph) the identifier of this response must be exactly

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      ID                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|QR|   OPCODE  |AA|TC|RD|RA|   Z    |   RCODE   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    QDCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ANCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    NSCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ARCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 2.5: *DNS header section format*

the same as the identifier of the query packet that initiated the response. This means the identifier has to be generated only when a query packet is generated, otherwise it should be a direct copy of the identifier of the query.

How these 16 bits should be supplied is not discussed in [45], neither is explained whether this is a signed or unsigned number. It is however good practice to fill it with 16 cryptographically random bits. (In the security section of this chapter this will be explained.)

Question or Response

The first bit following upon the identifier is the `QR`-bit. That stands for either a question or a response. When this bit is 0 the packet should be considered a query. If the bit is 1, the packet should be considered a response.

The four following bits are known as the operation code (`OPCODE`), and are always 0 in nowadays implementations, which means the query is a standard query. During design also 'inverse' and 'status' queries were considered, although none of these were actually implemented.

Authoritative

The bit after the `QR`-bit is the authoritative answer bit (or: `AA`-bit). This bit can only be set in responses. It means the response, and thus answer, can be considered authoritative. The server that responded the query states it is an authority for the requested query name (this will be explained later).

When looking back at the DNS topology illustration in Figure 2.4, a distinction can be made between which servers would give authoritative answers, and which not. If an `example.org` query is directed to the root name servers, this would *not* result in an authoritative answer response, as the root servers are not authoritative for the `example.org` domain. The root servers do however give a supporting answer in the form of NS-type answers that point to the `.org` name servers.

Sending the same question to the `.com` or `.nl` name servers would also not give authoritative answers, although this time also no NS-type answers will be responded. Directing the same query to the `.org` name server would give a NS-type response, that is however again not authoritative.

So the only time an authoritative answer is received, is when the query is addressed to the `example.org` name server. Responses received from this name server would therefore have the `AA`-bit set.

Notice this also implies that any recursing name server will *never* respond with the `AA`-bit set. As no caching name server will be authoritative for any domain, it only caches authoritative data, but when served to a client, this data will not be authoritative anymore.

Truncation

Directly after the `AA`-bit the `TC`-bit can be found. This truncation bit indicated whether a DNS packet is 'truncated' or not. Or, in other words: whether it exceeds the maximum DNS packet length. This maximum depends on what transport type has been chosen.

The DNS works over UDP as well as over TCP. And it uses port 53 in both protocols to be distinguished from other services. When a DNS packet is sent over UDP, the maximum packet length is set to 512 bytes in RFC 1035. Queries cannot exceed this limit, due to maximum domain name lengths, so only responses will have the truncation bit set. No maximum is specified by RFC 1035 when a DNS packet is transfered over TCP. However, because every DNS TCP packet is preceded by two bytes that define how long the DNS packet is, therefore a DNS TCP packet can be at most 65,535 bytes long.

So whenever the DNS packet exceeds one of both limits, the `TC`-bit will be set. When a UDP response is received with the `TC`-bit set, the client cannot fully rely on the response. It might suddenly stop, or contain half of the actual answer. Therefore it is urged to resend the query, but now over TCP, giving a much bigger response space.

Recursion – desired and/or available

In the previous section it was told what a DNS recursion (or: traversal) was. Usually authoritative name servers will not recurse, they will only answer to queries that are within their authority. Caching name servers on the other hand will recurse, meaning they will do all the work possible to answer a query, either positively (there is an answer), or negatively (there does not exist an answer).

The two bits following the `TC`-bit both have to do with this recursing feature. The `RD`-bit stands for recursion desired. With this bit set in a query packet, a client can state it wants the server to recurse for him. If the server supports recursion (i.e. is a caching name server) it responds with both the `RD`-bit and the `RA`-bit set. The `RA`-bit specifies the name server has recursion available, and can only be set in responses.

If a recursing name server allows recursion, but the `RD`-bit is not set in a query, the server will only respond authoritative answers.

The three following bits are known as the `Z`-bits, and are supposed to be 0 at all times. There is no specific purpose for these bits, however RFC 1035 states these are reserved for future use. RFC 2929 [33] – categorized as a 'best current practice' – clarifies statements made in RFC 1034 and RFC 1035, it also updates the purpose of some of the `Z`-bits. The last two `Z`-bits have found a destination nowadays, the DNSSEC chapter will introduce this new destination. Remark that more specifications have been updated in RFC 2929 (i.e. more `OP`- and `RCODE`s) although this section will concentrate on what is specified in RFC 1034 and RFC 1035.

Response Code

The next four bits represent the response code (or: `RCODE`). RFC 1035 states this field expresses the status of a certain response. Therefore all four bits should be zero when the packet is a query.

When the packet is a response on the other hand, several statuses can be formulated by the following decimal representation of the `RCODE` field:

0. No error;
1. Format error – For example when the query could not be parsed;
2. Server failure – There was a problem with the name server, therefore it was not able to process the query;
3. Name error – Sometimes referred to as `NXDOMAIN` responses, meaning a name server does not have data for this query available;
4. Not implemented – The name server did not implement the requested query;
5. Refused – The name server refuses to answer this query, this can have several reasons, for example only recursing is allowed for local clients;

6. – 15. Reserved for future use.

Counting

The most interesting and smallest parts of the header have been clarified by now. However, the remaining 8 bytes of the header have not been discussed yet. These 8 bytes are split into parts of 16-bits each. Each of these 16-bits values tell how many records are included in this packet. So besides the 12-byte header a DNS packet consists of more sections: the query section, the answer section, the authority section, and finally the additional section.

The first 16 bits number states how many questions the packet houses. However, every DNS packet nowadays includes only one question. So in nearly all cases the first 16-bits value represents a decimal 1. Notice that nonetheless whether the packet is a query or a response, it always includes one question in the question section. So when a packet is a response, it includes exactly the same query section as the query packet that initiated the request.

Next is the number of actual answers, listed as a 16-bits value. These answers correspond directly to the asked question in the query part of the query packet.

Further on there is the number of authority answers. These authority answers carry references to other authoritative name servers that might help the client further. For example, when asking about `example.org` to the root name servers, a root server will respond with a NS-type record pointing towards the `.org` name servers in the authority section of the response packet. Since it is not a corresponding (or: true) answer to the client's question, but it knows the `.org` name server might help the client further.

Finally the number of additional answers is stated. Additional answers are added by authoritative name servers to make life easier for clients. As these are regularly answers that might be helpful, according to the authoritative name servers answering the query. They will probably save the client some extra queries. An example hereof is a client looking up an MX-type record. (Such record type specifies what the Mail eXchanger for a certain domain name is.) A response to an MX-type query contains a domain name. This means that this query will be succeeded by an A- or AAAA-type query to resolve the actual address of this mail exchanger. A decent name server will put the MX-type record in the regular answer section, while in the additional section also the answer of the A- or AAAA-type query shows up – assuming the authoritative name server knows the answer to this question.

This principle of supplying additional information is also known as adding 'glue' (records) to the response. These glue records can become very useful when chicken and egg problems happen inside the DNS. An example of such problem is outlined like this: assume there is a domain name `example.net` that has two name servers (so two NS-type records): `ns1.example.net` and `ns2.example.net`. To do a DNS traversal, first the NS-type record of the `.net` is looked up inside the root name servers. Afterwards the `.net` name servers are contacted, to find out what the name servers of `example.net` are. This results in the expected response to contact either `ns1.example.net` or `ns2.example.net`. However, to contact any of these two name servers, the address of `ns1.example.net` should be known. This can be found by starting a new traversal, that begins exactly the same as the previous query. Only now ending up in a chicken and egg problem. Because first the address of `ns1.example.net` should be resolved, to find out what the A- or AAAA-type record of `example.net` itself is.

The solution to this problem is to let the `.net` name servers add two glue records, stating that `ns1.example.net` lives at `10.14.12.55`, and `ns2.example.net` at `10.12.1.54`. With this information a connection to one of both name servers can be made, to send the regular A- or AAAA-type query.

Remark that these additional records also bring security risks. This will be explained in the security section of this chapter.

Questions

As mentioned in the previous paragraph, a DNS packet generally houses only one question. The format of a question is portrayed in Figure 2.6.

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                                               /
/                     QNAME                     /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     QTYPE                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     QCLASS                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 2.6: *DNS question section format*

A question (or: `QNAME`) starts with an arbitrary (although maximum of 255) number of bytes. This name is split up into labels, that will have a maximum length of 63 bytes each. A label corresponds with a part of the domain string that is separated by dots. Such label is preceded by a byte that states the length of the label that is about to follow. Convention is that the highest two bits of this length byte are regularly 0 (that is where the maximum of 63 bytes for a label comes from). Notice a label is only allowed to consist of letters, digits, and hyphens. This means that not the entire character space of a byte can be used inside a label or domain name.

To illustrate how this labeling system works, an example of how the domain name `sample.example.org` will look like as a `QNAME` is given below.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|06| s| a| m| p| l| e|07| e| x| a| m| p| l| e|03| o| r| g|00|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

See how the domain name is ended by the null label. This null label can be seen as the label of the root.

Let's return to the question section format. After the `QNAME` it is stated what type the client is interested in (the `QTYPE`). This 16-bits number represents a query type.For example an A-type record has value 1, while the earlier mentioned MX-type record has value 15. Another special `QTYPE` is the `ANY`-type (value 255), this requests the name server to respond all records corresponding to the query name.

Next there is the 16-bits value that specifies in what class a client is looking for an answer (the `QCLASS`). This classing system was designed to support the usage of DNS in separate network protocols, that existed around the time of ARPANET. For example, at that time Chaosnet arose in the labs of MIT. Chaosnet had its own class inside the DNS: the `CH`-class (value 3). None of these networking protocols really gained popularity, nowadays almost only the Internet class (or: `IN`-class, value 1) is used. The `CH`-class is sometimes abused to propagate version information of name servers.

The `QCLASS` type also consists of an `ANY`-class. If a query is submitted with this class, the name server should respond with all known records that correspond to the query name, independent of the class.

Resource Records

By the counting paragraph it is known a DNS packet consists of several different sections. The previous paragraph explained what the question section looks like. The three remaining sections (answer-, authority-, and additional section) all consist of so called resource records (RR). A resource record can be compared with the 'record' term that has been introduced earlier. Note however that both terms will be used interchangeably in this thesis.

Resource records are answers to a question, and are dependent on the type of query to be answered. For example an A-type query demands a different answer than a MX-type query. Nevertheless, all resource records have the same format, which is illustrated in Figure 2.7.

A resource record starts with a response name, that is built up just like the query name inside the question section. So it consists of labels (that are prefixed with the label length) and the `NAME` is ended by the null label that represents the root. Notice the length of this name is variable, although it has a maximum of 255 characters and can only consist of letters, digits, and hyphens.

The `NAME` is followed by the 16-bits `TYPE` value that states what the type of this resource record is, succeeded by the 16-bits `CLASS` value that indicates the class of this record.

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                                               /
/                      NAME                     /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      TYPE                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     CLASS                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      TTL                      |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                   RDLENGTH                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                     RDATA                     /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 2.7: *DNS resource record format*

Next there is a 32-bits unsigned `TTL` value. The TTL has been discussed before and indicates to caching name servers how long the value can be cached, before the name server is urged to contact the authoritative name server again for a potential update of the resource record.

After the `TTL`, the `RDLENGTH` field can be found. This 16-bits unsigned value states how many bytes are to be expected in the `RDATA` section. This implies the `RDATA` section can only consist of 65,535 bytes.

The `RDATA` field is what can be called the actual value of the answer. The contents of this field are – as mentioned earlier – dependent on the type, and have an arbitrary length. Remark the `RDATA` field has no limitation on the usage of characters and such, and thus all 8-bits can be used to house data.

When looking at three different resource record types, a good indication can be given of how this field is filled.

A-type `RDATA` is in this case a 32-bits representation of the IPv4 address. This implicitly says that `RDLENGTH` will be 4, when the resource record has an A-type.

AAAA-type The `RDATA` section will store a 128-bits IPv6 address in network byte order. `RDLENGTH` will indicate 16 bytes of data is used.

MX-type The MX-type resource records are companioned with a priority number. This 16-bits integer precedes the mail exchanger's domain name in the `RDATA` field. Such domain name is built up like the query name. So it consists of labels and is ended by the null label. The total length of an MX-type record is the domain name length (including the null label), plus two bytes for the 16-bits number.

TXT-type This is the first time TXT-type records show up. A TXT-type record gives the DNS the ability to store arbitrary text information inside a resource record. The semantics of these texts are not specified by RFC 1035. Nowadays TXT-type records are used for email validation projects (i.e. SPF and DKIM) for example.

The format of the `RDATA` field is a collection of one or more character strings. Such character string is again preceded by a length byte, indicating a character string can have a maximal length of 255 bytes. Remark there is no null byte ending this sequence of bytes and there is no limitation on what characters are allowed in such string, the whole ASCII-set is allowed. In the DNSCurve chapter another usage of TXT-type resource records will be shown.

By ending the description of resource records, all sections of a DNS packet have been explained. The following section will focus on the security of the DNS protocol.

2.4 Security

This section focuses on security aspects of the DNS protocol. This knowledge is crucial in understanding later chapters that will discuss both DNSSEC (chapter 3) and DNSCurve (chapter 4). Large parts of this subsection are based on RFC 3833 [15]. First some essential knowledge is explained in the introduction subsection.

2.4.1 Introduction

While technical ins and outs of the DNS protocol have been introduced, nothing has been told about the network part of the DNS yet. This information is however essential to understand most of the security risks the DNS suffers, therefore this section starts with a small introduction.

The DNS is a stateless protocol. This means that communication between a server and a client is based only on packets that are received, and not on any (synchronized) data stream. A packet in the DNS means a series of bytes representing either a DNS query, or a DNS response.

The DNS uses UDP [53] as a building block for communication. However, a client will retransmit a query over TCP [55], when it received a response that was larger than 512 bytes. Later a new (DNS-related) RFC gives the ability to enlarge this limit, when discussing DNSSEC this will be further explained.

As mentioned, the ID field inside the DNS header identifies a DNS packet. This field can hold values up to 16-bits. This is however not the only value that identifies a DNS packet, name servers can also distinguish DNS packets by the source port of an arrived DNS packet. Since the source port size is specified by both the UDP and TCP specification, this is also 16-bits. So two 16-bits values will distinguish one DNS packet from another, that gives an identifying space of 32-bits.

It is possible to have more records for the same domain name and type. For example a domain name can have five different records, for the same query. Think of `example.org` having five different A-type responses: `10.2.3.1` up to `10.2.3.5`. These records with the same type and domain name are called ‘round-robin’ records. A client will pick one of such records as a final result. Notice the value of such records should indeed be different. This mechanism is usually used for (simple) load-balancing solutions.

2.4.2 Passive Attacks

Passive attacks are attacks that do not interfere with the regular operation of protocols. This is usually done by eavesdropping (or: sniffing) on a communication line.

One of the first things that comes to mind to a security aware person, is that all DNS packets are transmitted in the clear. There are no cryptographic measures taken to ensure an adversary would not be able to intercept any of the DNS communications. This makes intercepting DNS traffic easy, when an attacker is already able to eavesdrop on a communication line.

During such a passive attack, the adversary can become aware of privacy concerned data. Examples of such data are the usage of the communication line and even more dangerous: the whereabouts of users of this line on a certain network. Because every web browsing action is initiated by a DNS query, good insight in web usage of an organization can be collected.

2.4.3 Active Attacks

When looking at active attacks on a communication line, even more dangerous situations appear. If an attacker is able to both intercept and manipulate packets on the communication line (also known as ‘man in the middle’ attacks), a whole set of new attack vectors show up.

Being able to actively interfere with the communication, an A-type query for `www.example.org` can be detected, while the actual response will be blocked. The attacker will now insert a new DNS response packet with the same ID and addressed to the sender’s query source port, saying `www.example.org` lives at the web server of `evilpage.example.com` for example. Besides giving modified responses, also negative responses can be inserted, so that a client will think `www.example.org` does not exist.

An adversary could also block DNS queries or responses. In this way it can simulate that a server is unavailable. Or it can flip some bits of a DNS packet, leading to strange behavior of DNS implementations. All this leads to both availability and also integrity vulnerabilities.

Now that two general attack options have been discussed that are not specific to the DNS protocol, the next sections will discuss attacks that are only applicable to the DNS.

2.4.4 Passive Cache Poisoning

DNS packets consist of several sections. One of these sections is called the 'additional information' section. This section houses resource records that might be helpful for a client, according to an authoritative name server. Records in this additional section are usually referred to as 'glue records'.

Now take a client that is querying a AAAA-type record for the `www.example.org` domain. Assume an adversary controls this `example.org` domain, and its name servers. The authoritative name server for the `example.org` domain responds with a response DNS packet, that consists of one answer RR, two authority RRs, and three additional RRs. This response looks like the one portrayed in Figure 2.8.

```
;; QUESTION SECTION:
;www.example.org.             IN  AAAA

;; ANSWER SECTION:
www.example.org.    172800    IN  AAAA  fe80::1

;; AUTHORITY SECTION:
example.org.        172800    IN  NS    ns1.example.org.
example.org.        172800    IN  NS    ns2.example.org.

;; ADDITIONAL SECTION:
ns1.example.org.    172800    IN  A     10.2.3.4
ns2.example.org.    172800    IN  A     192.168.3.4
www.google.com.     2592000   IN  A     127.0.0.1
```

Figure 2.8: *Rogue DNS response including false glue records*

This is the first time this notation is used, and a brief explanation might be convenient. The four sections that follow after the packet's DNS header can be seen in the figure, prefixed by two semicolons. The question is obvious, and corresponds to what the text described. What is more interesting are the `ANSWER`, `AUTHORITY`, and `ADDITIONAL` sections. The resource records below the section names are formatted like this: the record's response name, followed by the record's TTL (in seconds), next the record's type, and finally the record's data. As mentioned before, the data is dependent on the record's type. For example A-type records have an IPv4 address as data, while NS-type records have domain names as data.

The authoritative name server of the attacker has included a correct response for both the answer-, and authority section. There is nothing special on the resource records in these sections. Also the first two RRs in the additional section seem okay. However, the last RR inside the additional section is the record that indicates something is not right.

If a caching name server would cache all received resource records including the ones in the additional section, it would also cache that `www.google.com` lives at `127.0.0.1`. Even for 30 days, as the TTL specifies. It is obvious this is absolutely not the way it should work. An authoritative name server that is in no way associated with the `google.com` domain, would now be able to make 'valid' statements about this `google.com` domain. In theory even about *any* domain name that exists.

This is not a working attack anymore nowadays, although it was until 1995. Around that time resolving name servers incorporated a mechanism that checks whether an authoritative name server is justified to include certain resource records. This check is known as the 'in-bailiwick' check. In short it means that resolving name servers will only accept resource records from the `.net` authoritative name servers that are in-bailiwick with the `.net` domain itself. Examples of response names that are in-bailiwick with the `.net` domain include `foo.example.net`, `on2it.net` and `subdomain.at.zeroxcool.net`. So resolving name servers are *not* allowed to use (or cache) resource records it received from `.net` authoritative name servers that are out-of-bailiwick. Examples of out-of-bailiwick domains for the `.net` domain include `w3.tue.nl`, `subdomain.example.eu`, `www.google.com`, and `example.org`. Notice however that an in-bailiwick check does not only apply to the additional (or: glue) section of a response, it applies to all sections.

2.4.5 Identifier Guessing and Query Prediction

Assume an attacker is not able to entirely control the communication line – although some influence is needed at the client side. In this case it will still be possible to interfere with the DNS functioning. A DNS packet is identified by two 16-bits values (i.e. the DNS packet's identifier and the client's source port).

An attacker would be able to forge a response by guessing both these 16-bits values. There are however more fields that should be 'guessed' correctly in order to let the DNS response packet appear to be valid. For example the response name(s) and type should both correspond to the query ones. It looks like a hard job for an adversary to do so, nonetheless the number of items that should be guessed correct can be reduced quite easily.

Imagine the attacker owns the website `www.verysecuresite.com`, containing hidden bogus image references to `www.example.org`. If a victim would visit this website, the attacker knows four things about the victim: the victim's IP address plus the query name, class, and type of the query the victim is about to ask. This is known because the victim has to lookup `www.example.org` to retrieve the (non-existing) images. This is the small influence an attacker should have to make a guess more likely to be correct. Now it should only guess the two 16-bits values correctly, resulting in a 32-bits probability space.

A few remarks must be made concerning the previous statement. In an ideal situation, there indeed is a true probability space of 32-bits. However, until recent – more on this in the next subsection – many software implementations did not make use of the total 32-bits probability space. Mainly because they used incrementing counters for the packet identifier. This makes it obviously easy for an attacker to predict the next identifier, if previous packets are known to the attacker. Besides, lots of implementations lacked the ability to alter the originating source port for every query. Some implementations even fixed the port their queries originated from. A combination of these flaws could sometimes lead to a probability space of nearly 1.

Even if DNS software would comply to use the entire probability space, soft- or hardware in the middle of transport can still influence a DNS conversation. There are for example known firewalls or NAT-applications that fail to originate DNS queries from different source ports.

Note that with this attack only A- or AAAA-type responses can be changed with rogue DNS data, because no other queries are made when a person visits a regular web page.

Now the attacker will build a response packet, including both the query name, class, and type, together with false DNS data. It will for example respond that `www.example.org` lives at `66.66.66.66` which is controlled by the attacker. To be even more effective, the TTL of this A-record should be reasonably high, so that it will be cached by the resolving name server for a long period. This all by itself is not enough. The artificial response packet should arrive before the real response packet arrives, and just as important: it should match the query identifier and source port of the initial query.

Sending only one of such forged packets gives a success probability of only 2^{-32} (assuming a true random number generator is used for both the identifier and source port), which is negligible. Therefore the attacker sends a flood of these artificial packets in a small period of time.

If a forged packet that has the right packet identifier, destination port, and all other essential corresponding fields, *and* arrives at the client before the real response packet arrives, the client will cache that `www.example.org` is located at `66.66.66.66`.

Applying such an attack at a resolving name server (instead of a single client) of course expands the impact. Recall however, this false DNS data is only served from the particular client that received the forged DNS response packet, and also for a limited period in time (usually the record's TTL).

Best defense against such attacks is to create a true probability space of 32-bits. This means every packet should have a cryptographically random identifier. And the query should be originated from a source port, that is also selected in a true random matter. Note there were also implementations that followed this advice, but used a bad (or flawed) random number generator. This would still make attackers able to predict identifiers and source ports relatively easy.

It should be noted that there are other (and easier) ways for an attacker to win this race against the legitimate packet. If an attacker is able to block any of the queries from a client to a name server, the server will never know a query existed and therefore not send a response. The other way around is also possible: if an attacker is able to block any of the responses sent by a name server, giving the attacker an opportunity to win the race much easier. Another way, that is the most easy one regarding the attack itself, is that the attacker is able to actively control the line. By sniffing it knows all details and is therefore able to generate a rogue response, and inject it before the legitimate response arrives.

2.4.6 Active Cache Poisoning

The previous attack is quite labor-intensive compared to the reached result – only one record that includes false DNS data. There is another type of attack that also makes use of guessing and query prediction, but has a much bigger impact. Due to in-bailiwick checks, adding bogus data to the additional record section is not of much help when the authoritative name server is not an authority for a certain domain. However, this attack requires the adversary to actively interfere with the DNS, by sending crafted packets. It is in that matter an attack that combines both the query prediction and the addition of rogue glue records.

As with the guess and query prediction attack, in this attack the client is also forced to query a certain domain name, in this case `001.example.org`. Note that any domain in `example.org`'s bailiwick would be okay too, only two requirements are that the domain should not exist, and a response should not already reside in a client's cache.

The correct response of the real authoritative name server would include a `NXDOMAIN` response code, meaning that `001.example.org` does not exist. Usually such a response is also equipped with resource records in the authority section- (stating the name servers of `example.org` are `ns1.example.org` and `ns2.example.org`) and additional section (denoting the IP addresses of the name servers, respectively `1.2.3.4` and `5.6.7.8`).

To see how the attack exactly works, the rogue DNS response packet that the adversary is about to flood the resolving name server with, looks like the one illustrated in Figure 2.9.

```
;; QUESTION SECTION:
;001.example.org.          IN  A

;; ANSWER SECTION:

;; AUTHORITY SECTION:
example.org.      172800   IN  NS    ns1.example.org.
example.org.      172800   IN  NS    ns2.example.org.

;; ADDITIONAL SECTION:
ns1.example.org.  2592000  IN  A     66.66.66.66
ns2.example.org.  2592000  IN  A     66.66.66.67
```

Figure 2.9: *Rogue DNS response actively poisoning a resolver*

This crafted DNS response looks a lot like the correct response like one of the real `example.org` name servers would give. There is no answer given – because there was none – and in the authority section the correct two name servers can be found. It gets interesting when a look is given at the additional section. These two records differ from the real response, and are in fact IP addresses of the adversary and the TTL is unreasonably high (30 days). Notice that both response names (`ns1.example.org` and `ns2.example.org`) are even in-bailiwick with the `example.org` domain, and will therefore be accepted by the in-bailiwick checks. When the caching name server would cache these additional section records, it means the attacker is in control of the entire `example.org`. Keep in mind that only the particular client will serve this false DNS data and only for a finite amount of time. This is usually the number of seconds the TTL prescribes, that is why the TTL was set so high by the attacker.

An advantage of this type of attack – besides the impact – is that whenever the insertion fails, for example because the real response arrived earlier than a correctly guessed forged packet, a new attack can take place immediately. This is done by letting the client retrieve `002.example.org` for example. If the query guessing attack would not succeed, another attempt could only be made if the TTL of the real response expired.

The idea of combining both attacks (identifier guessing plus query prediction, and rogue glue addition) was found mid 2008 by Dan Kaminsky [27]. It had quite some impact in the IT world, major hard- and software vendors quickly released patches that made the attack harder. These patches included cryptographically good random number generators – as source for the query identifier and source port – to increase the time period needed for this attack to be successful. Nevertheless, whatever patches are applied, this attack will always be possible using the regular DNS.

Long-term solutions on the other hand can be found in using DNSSEC or DNSCurve. Both protocols supply integrity to the DNS, making sure authenticity of DNS packets can be guaranteed.

2.4.7 Amplification Attack

An amplification attack is of a different order compared with the previously discussed attacks. This attack has nothing to do with the data the DNS handles, but specifically with the amount of traffic associated with this data, combined with source address altering.

As mentioned earlier, UDP [53] is one of the two transport protocols the DNS uses to transport its packets. Because UDP is of a stateless nature, it is possible to alter the source IP address in the underlying IP packet, without any hassles. UDP does not need a synchronized communication setup, like TCP needs. In this case a receiving host believes the UDP packet came from that altered source IP address. Hence also the server's response will be directed towards this modified IP address.

Now take the retrieval of the `www.example.org` domain name and assume there is a list of eight A-type round-robin records. A DNS query to the authoritative name server would in this case be 33 bytes big. A response to this query contains 161 bytes. This is an amplification factor of 4.87.

Indeed, 161 bytes is not a terrifying amount of data. But think of the fact a system with a 1Mbit/s[4] connection will fully use this bandwidth to send out this query packet, to several resolving name servers, and all packets originating from the same forged source IP address – of course assuming this source IP address is allowed to contact all these caching name servers. All name servers will send the response of the query to the forged IP address, resulting in a so called UDP flood of around 5Mbit/s in total.

Next consider this scenario, only now not initiated by one host, but in a networked approach by several hosts. For example, if 25 hosts would be equipped with a 1Mbit/s connection and use the same forged source IP address to query the caching name servers with, this would result in a flood of around 125Mbit/s. When the bandwidth of one link was used, this would only have caused a flood of 25Mbit/s.

It should be noted that although this is an attack specifically caused by the DNS, the consequences are not only limited to DNS hosts. The source IP address can be of any host, either running a DNS client or server, or nothing at all.

Solutions to stop or prevent amplification attacks are not bound to DNS

[4]Mbit = 1,000,000 bit

itself too. The actual problem is not caused by the DNS, since HTTP for example would have caused much higher amplification factors if it was used over UDP. Solutions should therefore come from preventing hosts to advertise spoofed IP addresses in UDP packets. A 'Best Current Practice' document [34] of the IETF introduces a plan for ISPs to filter all their outgoing traffic for non-owned source addresses. In this way an attacker will not be able to spoof IP addresses that are not owned by an ISP.

An amplification attack can be recognized by carefully monitoring the traffic of a name server. If this would be unreasonably high, detailed traffic analysis can give rise to what (spoofed) source IP addresses is causing the increase in traffic. An administrator could now decide to not respond to any new queries 'originating' from this IP address. An attacker could use this to cause a different availability problem: getting a large caching name server on a authoritative name server blacklist.

2.4.8 Hierarchical Trust

Something that is not really an attack, but good to keep in mind is hierarchical trust. As mentioned before, the DNS is hierarchy based. With hierarchical trust, also trustworthiness within the DNS is entangled.

A name server higher in the hierarchy enjoys more trust than a server lower in the hierarchy. This means `foo.example.org` should trust the `example.org`, the `.org`-, and the root name servers. If one of these name servers is not responding or gives for whatever reason wrong or flawed responses, chances are that `foo.example.org` will not be available. So taking care of a name server is one thing, trusting name servers higher in the hierarchy is something else.

Trustworthiness plays a big role in both DNSSEC and DNSCurve too.

2.4.9 Other Attacks

Until now mainly attacks have been discussed that specifically attacked the DNS protocol itself. With other attacks, attacks are meant that do affect the DNS, but are not caused by the DNS itself.

Examples hereof are DNS hosts that have been compromised, or name server software itself that contains buffer overflows leading to a compromise. Due to the hierarchic nature of the DNS, this indicates that name servers higher in the hierarchy should be considered more likely to be a target for attackers. This implies these hosts should be secured with more care.

Until now only the amplification attack involved availability aspects. This was discussed since the DNS could be used to initiate availability attacks.

But what if name servers themselves are under any denial of service (or: DoS) attack – such as UDP floods. Or what if a name server needs hardware maintenance. This would mean no DNS client would be able to get a response to a query. As already mentioned in the beginning of this chapter, the DNS is designed with availability in mind. If one name server stated in NS-type resource records would be unreachable (meaning no response is received within a particular time interval) another one is contacted. Although if all of the authoritative name servers of a domain would be unavailable, no responses could be retrieved by a caching name server. This leads to availability issues. Like other availability problems in networking (mainly DoS attacks), this is very hard to be fixed.

2.4.10 Summary

In this subsection all attacks that have been discussed will be listed. Table 2.1 illustrates this summary. All shown attacks are listed, together with what component they attack (i.e. **c**onfidentiality, **i**ntegrity, **a**vailability, or a combination thereof). Next, it is also shown whether the attack is still possible and an explanation why.

CIA	Name	Possible?	Why?
C	Passive Attacks	✔	Possible because everything is sent in plaintext.
CIA	Active Attacks	✔	Possible because no integrity and authenticity constraints are applied to the DNS protocol. Attacks all facets, since attacker has entire control over the communication line.
I	Passive Cache Poisoning	✘	Not possible anymore due to in-bailiwick checking.
I	Identifier Guessing and Query Prediction	✔	Possible because no integrity and authenticity measures are taken. Can however be slowed down by using cryptographic random packet identifiers and source ports.
I	Active Cache Poisoning	✔	See above.
A	Amplification Attack	✔ / ✘	Possible, although not really effective when looking at achieved amplification factors.
I	Hierarchical Trust	✔	Not really an attack, but a reminder that zones higher in the hierarchy have more responsibility when it comes to trust. I.e. they implicitly control all zones beneath them.
CIA	Other Attacks	✔	Attacks that are out of the scope of the DNS are still possible, plus they can attack all facets.

Table 2.1: *Summary of all attacks on the DNS including their effectiveness*

Chapter 3

DNS Security Extensions (DNSSEC)

In the previous chapter the regular Domain Name System has been introduced with all its ins- and outs. When looking at security issues introduced by the DNS, quite some parts of the protocol specification appear to be wrong by design. The Domain Name System Security Extensions (abbreviated to DNSSEC) have been introduced to solve some of these security issues. This chapter will discuss the general workings of DNSSEC, but will not go into so much detail as the DNS chapter did. This master project mainly focuses on DNSCurve. However, to understand the differences between both systems, a good introduction to what DNSSEC is and does, is needed.

This chapter starts with a general introduction into DNSSEC and its terminology. The history of DNSSEC is discussed in the next section. Next, the general objectives of DNSSEC will be explained. Afterwards, more detail is given on the general working of DNSSEC. In that section also a DNSSEC traversal will be shown. Finally, this chapter will tell what security issues have been mitigated by DNSSEC, and what not.

3.1 Introduction

The evolution of DNSSEC is a story on its own. Therefore a short walk through time is given in the next section. In this section however, a brief introduction is given to what DNSSEC is, and what it does. Afterwards there is a part destined to introduce new terminology.

What DNSSEC is, and what it does; is answered in one sentence by [12]: “The DNS security extensions provide origin authentication and integrity protection for DNS data, as well as a means of public key distribution”. The unofficial DNSSEC website [8], that tries to be a central point in documenting DNSSEC sources, states: “It is a set of extensions to DNS, which provide: a) origin authentication of DNS data, b) data integrity, and c) authenticated denial of existence". An important point that is missing in this description is that DNSSEC should be backwards compatible with the regular DNS. This would make deployment much easier, because organizations can gently move on to DNSSEC.

3.1.1 Terminology

To introduce the general working of DNSSEC, it is good to understand some more terminology regarding the traditional DNS.

Domains and Zones

The term 'domain' has already been introduced, and used quite a lot in this thesis. Although a formal definition has not been given yet: a domain is the complete subtree of the DNS tree, which is rooted by the lowest node in the hierarchy the domain describes. To get a better understanding of this definition, a renewed domain map of Figure 2.2 is shown in Figure 3.1. The entire orange colored area in the figure illustrates the `example.org` domain.

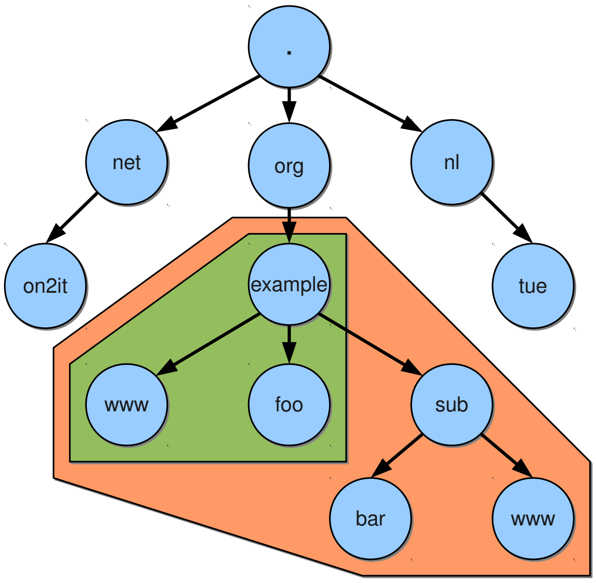

Figure 3.1: *Domain tree map showing a domain and a zone*

A term that has not been brought up yet – but that is important in DNSSEC – is that of a 'zone'. A zone is closely related to a domain, however it is dependent on the authority of a name server. In short: a zone is the subtree of the DNS tree although only including nodes for which the name server is an authority. References (or: delegations) of parts of this DNS tree do therefore not count as being in the same zone. Now assume there is an authoritative name server at `10.2.3.4`, that is an authority for the `example.org` domain. However `sub.example.org` has been delegated to the `192.168.14.12` name server. This means the `example.org` zone only consists of the `example.org`, `www.example.org`, and `foo.example.org` names. In Figure 3.1 this is shown by the green colored area.

Another zone in this figure is one consisting of `sub.example.org`, and its children `bar.sub.example.org` and `www.sub.example.org`. That are all served by the `192.168.14.12` name server. While all other non-mentioned nodes are zones on their own.

Resource Record Sets

Resource records (RRs) already have been introduced in the previous DNS chapter. A resource record consists of a response name, a record type, a class, a TTL value, and finally the record's content (or: value). Resource record sets (RRsets) on the other hand have not been discussed yet. They are – as the name indicates – collections of resource records. Although not every two resource records can be considered an RRset.

RRs can be called an RRset when they have exactly the same response name, record type, class value, and TTL[1]. What makes them distinguishable of each other is therefore the content. Figure 3.2 portrays which two records can be called an RRset, and which not. Remark RRsets can consist of more than two records, and a single resource record can be called an RRset too.

```
example.org.   3600   IN   A    127.0.0.1   | RRset
example.org.   3600   IN   A    127.0.0.2   |

example.org.   3600   IN   A    127.0.0.1   | not an RRset
example.org.   3600   CH   A    127.0.0.2   | classes differ

example.org.   3600   IN   A    127.0.0.3   | not an RRset
example.org.   3600   IN   AAAA ::1         | types differ

example.org.   3600   IN   A    127.0.0.4   | not an RRset
example.net.   3600   IN   A    127.0.0.5   | response names differ

example.org.   3600   IN   A    127.0.0.1   | not an RRset
example.org.   7200   IN   A    127.0.0.2   | TTLs differ
```

Figure 3.2: *Example of what can be called an RRset and what not*

[1]There is some unclarity whether the TTL should be equal to the TTL of other resource records inside a resource record set. For DNSSEC this is the case, according to RFC 4034 [13].

3.2 History

The regular DNS was introduced in the beginning of the 1980s and became a standard in 1987 [44] [45]. At that time the Internet evolved towards the network as it is known nowadays [37]. Not only governmental organizations, research institutes, and universities situated in the U.S. were connected to the Internet, but also non-U.S. based hosts were able to communicate with each other. When the security problems the DNS suffers are considered in this historical perspective, it can be concluded that during its design not enough measures were taken to prevent these attacks.

In the beginning of the 1990s the first vulnerabilities of the DNS protocol came to light. Steven Bellovin wrote a paper [16] describing an attack. This paper was however withheld from publication until 1995 – “because it described a serious vulnerability for which there was no feasible fix”. This publication did not only warn organizations using the DNS, but also officials in the U.S. government. As the Internet became more and more important for communication within the government itself – in particular for the Department of Defense – it was decided at the end of 1995 by the IETF to start working on a new or at least updated DNS protocol. The IETF dedicated this work to the DNSEXT working group, that was setup to “actively advance DNS protocol-related RFCs on the standards track while thoroughly reviewing further proposed extensions” [1]. This working group started specifying what security guarantees the new DNS had to meet. During initial meetings of the DNSEXT it was decided to focus on integrity of the DNS.

A first attempt to provide a secure DNS was specified in RFC 2065, that is titled: ‘Domain Name System Security Extensions’ and published at the beginning of 1997. This is the first time the name DNSSEC pops up. Remark this is two years after the DNSEXT working group was asked to work on a secure replacement.

When people started implementing RFC 2065, several problems and unclarity arose [47]. Leading to a new RFC: RFC 2535, this RFC obsoleted RFC 2065 although it carried the same title. RFC 2535 was published at the beginning of 1999 and did this time lead to a working implementation at the end of ’99, developed by the Internet Systems Consortium (ISC). ISC implemented DNSSEC support for their BIND name server implementation.

However, there was a working implementation at that time and a – at that moment apparently – ready RFC, massive deployment of DNSSEC did not start. In 2001 experiments and also real world cases showed that RFC 2535’s key handling caused “operational problems” [47]. Leading to a re-factor of the DNSSEC protocol, meaning the process of writing, drafting, and publishing was started again. The documents related to this new version are being referred to as RFC 2535-*bis*. The team changed their way of working, by correspondingly publishing a new snapshot of the BIND name server, that was capable of handling the 2535-*bis* protocol.

Eventually in 2005 three new RFCs were published, that obsoleted almost every previously released DNSSEC related RFCs. DNSSEC as specified by these three RFCs is also referred to as DNSSEC-*bis*. RFC 4033 [12] introduces DNSSEC in general, RFC 4034 [13] discusses the newly introduced DNSSEC record types, while RFC 4035 [14] explain the actual changes to the protocol. This chapter focuses only on these last three RFCs, because details specified in the obsoleted RFCs are not in use anymore.

After the release of the new RFCs, the `.se` domain was the first ccTLD that publicly deployed a DNSSEC environment at the end of 2005. Recent activities regarding DNSSEC are the actual deployment of DNSSEC on the root servers. At the beginning of May 2010 all root servers have been made ready to serve the signed root.

On June 16, 2010 the first key signing ceremony was held in Culpeper, Virginia, US. At this ceremony parts of the key signing key (KSK) for the root zone have been generated. On July 11, 2010 the second part of the key signing ceremony was held. This time in El Segundo, California, US. The root zone trust anchors (i.e. the actual keys) of the signed root zones have been published on July 15, 2010, making the signed root truly available and DNSSEC fully deployed at the root servers. This is an important date, mainly due to the hierarchic nature of DNSSEC. This will become more clear later on in this chapter.

3.3 Objectives

This section will explain the objectives of DNSSEC. And in contrast to that – maybe even just as important – also the non-objectives of DNSSEC, i.e. what DNSSEC mentions it does not solve.

In general DNSSEC was introduced to protect the regular DNS for known attacks. It is however good to observe that DNSSEC was not designed to *prevent* all known attacks, but to be able to *detect* them. The introduction of this chapter already discussed the three objectives of DNSSEC. There [8] stated the objectives in one sentence: “a) origin authentication of DNS data, b) data integrity, and c) authenticated denial of existence”. Another one was added in this thesis: “d) backwards compatibility with regular DNS”. Let’s discuss each of these objectives in detail.

3.3.1 Origin Authentication of DNS data

When one addresses a letter to someone, the letter usually includes the sender’s signature. This gives the recipient of the letter a small proof that the letter really was sent by the one who’s name is at the bottom of the letter.

Looking at the DNS world, such proof would be of great value. Because a client can trust the authoritative name server it really is an authority for a certain zone[2]. In the security world this principle ('knowing the data truly came from the source it claims it originated form') is called 'authenticity', and is also seen as a special case of integrity.

3.3.2 Data Integrity

Returning to the letter example, it is possible that someone in the middle of transporting the letter, from sender to recipient, is able to rewrite some sentences. Or to hide lines, or to add sentences. Such alterations can be disastrous.

When this example is aligned with the DNS, things are just as bad. Anyone, or anything in between a DNS 'conversation' can alter a DNS query or response without the recipient knowing. DNSSEC should therefore be able to detect unauthorized modifications. This phenomenon is called maintaining the 'integrity' of the data.

3.3.3 Authenticated Denial of Existence

During the discussion of DNS in the previous chapter, the focus mainly lay on positive answers to DNS queries. Negative answers are however just as important. Sending a query for a domain name to an authoritative name server is one, but knowing whether there *is* an answer is two.

Some detail has been given on 'denial of existence' in the technical section of the previous chapter. A negative DNS response is equipped with the same query name in the packet, but with a response code (`RCODE`) stating this is a non-existent domain (`NXDOMAIN`). However, just like positive responses that are being forged, negative responses are vulnerable to this attack too. It is therefore DNSSEC's job to be able to detect such rogue denial of existence responses.

3.3.4 Backwards Compatibility with Regular DNS

It is obvious that a transition from the regular DNS to a DNSSEC environment in one click is highly unlikely. Therefore, DNSSEC should be backwards compatible with the regular DNS. So that hosts that do not support DNSSEC yet, are still able to use the old DNS protocol. Nevertheless, by not using DNSSEC these hosts will not benefit any of the DNSSEC security guarantees.

[2]It should be noted this is not shown by the authoritative name server itself, but by a name server higher in the hierarchy.

3.3.5 Non-Objectives

During the initial design process of DNSSEC, directly some influential decisions have been made, like what DNSSEC will not do. In the paragraphs below these statements will be discussed.

Confidentiality

It was a “deliberate design choice” [12] not to make confidentiality an objective of DNSSEC. This means all DNS traffic will still be sent in plaintext. Implications of this design choice will be given in the security section of this chapter.

Availability

DNSSEC does not protect against any threats the name server itself would be vulnerable for, without running DNSSEC. Threats of this kind are for example buffer overflows in the name server software, or denial of service attacks targeting the name server. Although in the security section it will be shown that DNSSEC makes things even worse, regarding availability.

Others

There are also other kind of objectives that DNSSEC does not solve, although not explicitly mentioned by RFC 4033, one of which being the guaranteed correctness of the data. If a DNS data manager makes a mistake, for example to point NS-type records to the wrong name server, DNSSEC will not cover such errors. It will however ensure to a client this wrong response is really coming from the authoritative name server it addressed the question to. This means DNSSEC will not guarantee correctness of data, it will only guarantee authenticity of the data.

Following on this, DNSSEC will also not solve any so called typosquatting. This phenomenon focuses on typographic errors people make. Instead of visiting `www.google.com`, people misspell and visit `www.goolge.com`. Notice these ‘attacks’ usually go hand in hand with phishing attacks. For example because the website behind the misspelled name contains mal- or spyware. Obviously DNSSEC does not protect against such ‘spelling error’ attacks.

3.4 Specification

Now that the regular DNS protocol is introduced, together with some new terminology that is important for DNSSEC, it is time to start explaining DNSSEC itself. This section will do that, starting by introducing the new resource records DNSSEC defined. The next subsection describes some of the cryptographic primitives that DNSSEC offers. Followed by a subsection that focuses on key usage inside the DNSSEC environment. Next, a DNSSEC traversal – that is already known from the regular DNS chapter – will be shown. This should make the earlier presented material much more evident. Afterwards the topology – also known from the regular DNS chapter – of a DNSSEC situation is explained. Finally, implications for data managers are discussed.

3.4.1 Resource Records

To comply to all objectives listed before, DNSSEC introduced new resource record types in RFC 4034 [13]. In this subsection all of these new types are discussed. Like mentioned in this chapter's introduction, no very detailed information on the technical protocol itself is given. For example, wire data formats of the new resource records have been left out.

DNSKEY

DNSSEC uses public key cryptography to sign and authenticate data. Public key cryptography needs an input and a key. In the regular DNS there is no standardized way to represent key material in. Of course there is the TXT-type record that can house text, but semantically it is not preferred to store key material in such records.

Therefore the DNSKEY-type resource record has been introduced by RFC 4034, to be able store public keys. However, the RFC states in particular that DNSKEY-type records are "not intended as a record for storing arbitrary public keys (...) that do not directly relate to the DNS infrastructure".

An interesting field of the DNSKEY-type resource record is the `protocol` field, that specifies for what protocol this key is destined for. At the moment RFC 4034 only allows this to be DNSSEC. Then there is the `algorithm` field, that determines for what algorithm type this key is used for. Examples of these are the RSA/SHA-1 combination, or the DSA/SHA-1 combination. Some more detail will be given on these cryptographic algorithms in the next subsection.

Finally the resource record of course exists of the key material itself. The length of the key and the way it is stored is fully dependent on the algorithm that is used.

RRSIG

When a piece of data has been signed by using someone's private key, this signed data is then called a 'signature'. In the regular DNS there is no way of representing signatures in DNS records, that is why the RRSIG-type record has been introduced. Likewise, RFC 4034 specifies that RRSIG-type records should only be used to store signatures that validate DNS record values.

The following fields of a RRSIG-type record are worth mentioning. The `type covered` field specifies for what RRset-type this RRSIG record supplies a signature for. So notice a signature is generated on the entire RRset, not on a single record (or the RRset would consist of only one record of course). Next, there is the `algorithm` field that is already discussed in the DNSKEY description, and prescribes what algorithm is used to generate the provided signature. The `original TTL` field does exactly what is states: it provides the TTL of the RRset this record represents a signature for. Remark that this implies all TTLs inside an RRset should be equal.

Next, the `signature expiration`, and `signature inception` fields can be found. These express the validity period of the particular signature. Next there is the `key tag` field. Key tags are being used to efficiently pick a DNSKEY-type record from a set of key records, they are however not identifying. The `signer's name` field specifies in what zone the DNSKEY-type record can be found, that has been used to generate this signature. And finally there is the `signature` field itself.

The exact signature is worth to look at. RFC 4034 specifies that the signature is generated from the following string: `RRSIG_RDATA | RR(1) | RR(2) | ...`[3]. Where `RRSIG_RDATA` is the string of all the previously discussed fields, except for the `signature` itself, of course. The resource records RR(i) are elements of the resource record set. A single resource record consists of all the strings that specify the resource record (such as the response name, type, class, TTL, content length, and the actual content/value).

Remark that during the description of a DNSSEC traversal, many of these fields will become more clear.

NSEC

One of the objectives of DNSSEC is to be able to authenticate negative responses. To accommodate this, the NSEC-type record has been introduced. The name NSEC has been derived from Next SECure record, which exactly describes its function.

Assume the following zone information exists at an authoritative name server: `alfa.org`, `charlie.org`, and `delta.org`. Now a client queries this name server for `bravo.org`. This domain name is not part of the zone the server serves, so in the standard DNS a `NXDOMAIN` response will be returned. This will still be done by DNSSEC aware name servers, however no signature will be generated in realtime for responses like these – why exactly will be explained later.

[3]The `|` symbol represents a concatenation.

Therefore, a DNSSEC aware name server will respond with the NSEC-type record `alfa.org. IN NSEC charlie.org.`, indicating there are no records between `alfa.org` and `charlie.org`, lexicographically seen. So NSEC-type records describe the 'gap' between response names.

Let's look at most interesting fields of a NSEC-type record. First there is the `next domain name` field, that specifies what the next domain name is, that follows after the resource record's response name. Next, there is the `type bit maps` field. This specifies what RRset types there exist, having the owner's name response name. So if Figure 3.3 would be a part of the previously discussed `.org` zone, it means the type bit map of the NSEC-type record specifies that the `alfa.org` zone consists of NS-, RRSIG-, and NSEC-type RRsets. Which is exactly what the NSEC record type in Figure 3.3 tells.

```
alfa.org.     86400  IN  NS     a.ns.alfa.org.
alfa.org.     86400  IN  NS     b.ns.alfa.org.
alfa.org.     86400  IN  RRSIG  NS 5 2 ...
alfa.org.     86400  IN  NSEC   charlie.org. NS RRSIG NSEC
alfa.org.     86400  IN  RRSIG  NSEC 5 2 ...

charlie.org.  43200  IN  NS     ns1.charlie.org.
...
```

Figure 3.3: *Part of the `.org` zone showing denial of existence records*

It can already be seen these NSEC-type records 'leak' interesting information the regular DNS would never publish, if it was not asked for it. With a trivial algorithm the entire `.org` zone could be determined. This phenomenon is called NSEC walking.

Although DNS information itself is relatively public – i.e. everyone can query for names under a certain domain – knowing the entire zone of a domain is unwanted behavior. Because detailed network information can be determined from this information. Some data managers even store hard- and software information inside their DNS.

This lead to the introduction of yet another new RFC, that specifies a new record type called NSEC3.Later this new NSEC3-type record will be briefly discussed.

DS

In the history section of this chapter, it could be seen that quite some refactors of the DNSSEC protocol have been made. One of these refactors was caused by what was earlier noted as "operational problems". These operational problems were due to RFC 2565's key management. It is clear by now signatures played (and play) a huge role. In the old DNSSEC, changing a

DNSKEY-type record somewhere down in the tree, would result in lots of re-signaturing in all layers higher in the hierarchy of this record.

If an organization would lose its private key part, or when the key is compromised, all child zones should retransmit their key records to be resigned by the parent zone. Imagine that a zone has millions of child zones (like the `.com` zone for example), this would result in a huge administrative process.

The main thing that has been changed in DNSSEC-*bis* is the introduction of the DS-type record. The 'Delegation Signer' record makes the process of key rollovers much easier. Because actions taken after such change only require attention at one layer up in the hierarchy.

A delegation signer record is mainly a reference to a DNSKEY-type record down in the hierarchy. Although reference is not the right word, since a DS-type record contains a cryptographic hash of the key material. Because this DS-type record will be signed at the parent (by using a RRSIG-type record), it in fact states that the key down in the hierarchy can be trusted.

Looking at the contents of a DS-type record, the following interesting fields can be distinguished. First there is the `key tag`, that is already known from the discussion of the RRSIG-type resource record. A `key tag` is a non-identifiable reference to a DNSKEY-type record, and is generated from all fields of a specific DNSKEY-type record. This `key tag` corresponds to the `key tag` of the DNSKEY record this DS-type record is making a trusted delegation for. Next, there is the `algorithm` field, that directly correlates to the one used in the DNSKEY record. The `digest type` field indicates what digesting algorithm is used, for the next field: the `digest` field. This field houses the output of a cryptographic hash function. The hash function is fed with the response name of the DNSKEY-type record, and all the fields of the DNSKEY record where this DS-type record is delegating to.

Again, lots of these details will become clear when a DNSSEC traversal is being discussed.

NSEC3

As mentioned, NSEC3 is an alternative to the NSEC-type. It is specified in RFC 5155 [42] – which is fully dedicated for this purpose – and published at the beginning of 2008. Indeed, around 3 years after the initial DNSSEC RFCs.

Just like NSEC-type records, it provides authenticated denial of existence in DNSSEC. Only it solves the so called NSEC walking. To accomplish anonymity for the gaps in between response names, the NSEC3 record type 'hides' this information by using cryptographic hashes. This prevents clients from direct online walking through records. This can however still be done in certain situations in an offline fashion, the security section of this chapter will go into more detail.

Another element that NSEC3 improves is the possible minimization of the signing process for zones that like to make use of this feature. In the data manager subsection of this section, it will become clear that zone sizes will grow linear in size when they have been signed. Even a small modification results in quite some resignaturing. For very large zones, such as `.com`, this is an operational problem. This is one of the reasons why the `Opt-Out` flag was introduced in NSEC3.

DNSSEC will not sign data it is not authoritative for, why exactly will be explained later. This means that for example NS-type delegations will not be signed, since they are outside the zone's authority. However if a zone is DNSSEC capable, a DS-type delegation record should be made that is also out of the current zone's authority, therefore DS-type records are an exception to this rule. So whenever a zone is not DNSSEC capable, a denial of existence record should exist stating that there is no DS-type RR between the delegation zone and a next zone. Of course this record should be accompanied by a signature too.

The minimization that NSEC3 introduces, has to do with withholding unsigned records. If the `Opt-Out` flag is set, this means that in between the gap unsigned delegations can exist, i.e. delegations that do not have a DS-type record. In this way, these unsigned delegations do not need a separate NSEC3 record, plus corresponding RRSIG. Decreasing the signed zone's size. If the `Opt-Out` flag is not set, this means the zone is `Opt-In`. Meaning it behaves just like the 'old' NSEC in this manner, i.e. supplying denial of existence records for all RRs, even non-authoritative ones.

3.4.2 Cryptographic Primitives

It will be clear by now DNSSEC is built upon cryptographic primitives. In this subsection all used primitives in the DNSSEC world will be discussed.

History

The first DNSSEC RFC (i.e. RFC 2065) already had the possibility to support more cipher suites, although it only allocated one algorithm number. Algorithm number one was reserved for the RSA/MD5 cipher suite. This combination was at that moment (1997) widely used, and specified in the so called PKCS #1 document [57]. As key size, the RFC recommends 640 bits, at that time already a key size that proved not to be very secure. However, the RFC states that "high level zones in the DNS tree may wish to set a higher minimum, perhaps 1000 bits". A defined maximum key size was specified at 2552 bits.

The next RFC in the DNSSEC life cycle (i.e. RFC 2535) also had the ability to support several cipher suites. Only this time more standard algorithms have been specified. To be backwards compatible, algorithm number one has been reserved for the RSA/MD5 combination. It also allowed to use some new primitives, such as DSA and "elliptic curve crypto". RFC 2535 expressed that implementation of DSA is mandatory, while the RSA/MD5 received a 'recommended' implementation advice.

Algorithm Types

Let's focus on the current DNSSEC RFCs. RFC 4034 [13] specifies all algorithms and digest types in its first appendix. All algorithms that can be used are portrayed in Table 3.1. (Note that the usage column will be discussed later.) Later on RFC 5155 [42] added two alias algorithms (algorithms #6 and #7) to the specification. These aliases are no new algorithms, but add flags to let clients recognize that the particular zone uses NSEC3-type records for denial of existence, instead of the old NSEC-type records. Next, also RFC 5702 introduces two new combinations, replacing SHA-1 by two of its successors. Both new hashing algorithms are considered more secure, and better collision resistant.

#	Algorithm	For DNSSEC	Status	Usage
1	RSA/MD5	No	Not recommended	0.02%
2	Diffie-Hellman	No	–	0.00%
3	DSA/SHA-1	Yes	Optional	0.19%
4	Elliptic Curve	–	–	0.00%
5	RSA/SHA-1	Yes	Mandatory	95.56%
6	DSA/SHA-1–NSEC3/SHA-1	Yes	Optional	0.00%
7	RSA/SHA-1–NSEC3/SHA-1	Yes	Optional	4.04%
8	RSA/SHA-256	Yes	Optional	0.24%
10	RSA/SHA-512	Yes	Optional	0.06%

Table 3.1: *DNSSEC algorithm types*

As can be seen, not all algorithms can be used in DNSSEC itself. The algorithms marked to be used with DNSSEC are allowed as an algorithm number in the `algorithm` fields of DNSKEY-, RRSIG-, and DS-type resource records. The other algorithms are not allowed to be used by DNSSEC and are mainly included for backwards compatibility.

Algorithm #4 is open for discussion, at this point no algorithm set is specified for elliptic curve crypto. Any DNSSEC implementor is forced to support the RSA/SHA-1 combination in its implementation. All other combinations are optional and it is up to the implementor whether these will be supported by the implementation.

Digest Types

Cryptographic digests (or: cryptographic hash functions) are used in two different ways in DNSSEC. One way is the one that is discussed in the previous paragraph. These hash functions are used to shorten the amount of data to be signed, namely only the fixed size hash the digest function generated. The choice for such digest function is already made. Almost all combinations of the previous paragraph use SHA-1 for that.

Another way digests are used inside DNSSEC is to represent DNSKEY-type resource records into a fixed size string. What cryptographic hash function is being used for this purpose, is determined by the DS-type resource record. The `digest type` field indicates what function of Table 3.2 is used. RFC 4034 only specifies the first algorithm, later on RFC 4509 added the second one. Nowadays protocol implementors are obligated to implement support for both algorithms.

#	Algorithm	Status
1	SHA-1	Mandatory
2	SHA-256	Mandatory

Table 3.2: *DNSSEC digest types*

Usage

It is good to look at what primitive combinations there are currently being used in the DNSSEC environment. And even more interesting: what primitives are mostly used. SecSpider [51] – a research project of UCLA – gives good insight in these questions.

At the moment of writing there are almost 25,000 DNSSEC enabled zones. More than 23,000 of these zones can be considered non-testing zones. These production zones, use over 58,000 different keys. Indeed, there are more than 2 times the number of keys, as there are zones. This is caused by special key management, that makes a distinction between keys. More on this is explained in the next subsection, that focuses on key usage.

The differentiation between the algorithm types that are used, is already shown in Table 3.1's last column. The RSA/SHA-1 combo is used the most. The reason is obvious: every implementor is forced to support this algorithm type. There is however a trend visible that more zones (and thus keys) are NSEC3 ready, explaining the rise of algorithm #7.

3.4.3 Key Usage

The current DNSSEC RFCs make no explicit distinction between keys. However, to make key management easier RFC 4641 [40] introduces some best practices. One of which is to make a distinction between key types.

It is known from the RRSIG-type record explanation, signatures have a validity period. In this period a signature is considered valid. Although not explicitly included as a field somewhere, cryptographic keys usually also have a validity period. This all relies on one point: in what period can an attacker determine valuable information regarding the used key. The time in which key information can be determined, and a new key should be rolled out, is dependent on several factors. For example the type and algorithm the key is generated for, but also the key length is an important factor. In general keys with a larger key length can have a longer validity period.

This key validity period is so important for DNSSEC, because of the distributed nature of the DNS. Although at this point a DNSSEC traversal is not fully clear yet, it is known the DS-type resource records point to DNSKEY-type records in the same zone or in child zones, but not in zones that are higher in the hierarchy. The meaning of DS-type records can be put like this: 'trust in the zone that is in my response name is built upon the DNSKEY record I am pointing at'. Notice a DS-type record, and the DNSKEY are usually located at different zones. But there is an obvious connection between the two. The DS-type record contains a digest of the DNSKEY that lives in a lower (or the same) zone. This indicates both zones need to communicate, to get possession of this digest.

Now assume these zones are also physically split, i.e. the DS-type record is served by another name server than the DNSKEY-type record is. If the zone in which the DNSKEY-type record lives would roll over a new key – because according to its key lifetime procedures the security of the key cannot be guaranteed anymore – this means this new key should be retransmitted to the name server that is serving the DS-type record, and resigned there. This sounds like a doable operation, but assume the zone serving the DS-type record is the `.com` zone (consisting of more than 87 million child zones), and every child zone updates its DNSKEY-type record once a month. This means the `.com` authoritative name servers would have to sign 33 DS-type records per second, on average. Maximizing the period in which a key should be rolled over would definitely help decreasing the number of sign operations. From the previous subsection it is known the choice of the algorithm is already implicitly made. This leaves only variety in the key length to enlarge the time period.

However, choosing a large key size has other influences besides bigger DNSKEY-type records. Also RRSIG-type records will grow in size, plus signing and validating the signatures will take longer. It probably becomes clear why two distinct key types have been introduced.

Keys belonging to the first type are called KSKs (Key Signing Key)[4]. These keys will authenticate other keys. This is done by signing these keys using the KSK. KSKs themselves are being authenticated by the DS-type resource records that are housed higher in (or at the same level of) the hierarchy. To minimize the communication between parental and children zones, long key lifetime periods are preferred. This means that KSKs usually have a long lifespan, directly implying a large key size. RFC 4164 states the following guidelines: “1024 bits for low-value domains, 1300 bits for medium-value domains, and 2048 bits for high-value domains”. This quote does indeed not explicitly mention KSKs. Nevertheless, because KSKs control an entire zone’s trust, KSKs should be considered medium-, or high-valued ‘domains’. A large key size for KSKs is not necessarily bad, because it will only sign a small portion of a zone, namely the so called ZSKs (Zone Signing Key).

These ZSKs are being authenticated by the KSKs, and they authenticate zone data themselves. Zone data authenticated by ZSKs can be considered low- or medium-valued domains. Implying smaller key lengths for ZSKs. If a ZSK would be rolled over, no higher level zones have to be contacted, as the new zone key will be signed by the KSK, that is locally available in the zone. Remark that KSKs (i.e. the DNSKEY-type records representing the KSK) are usually located in the same zone as the ZSKs they authenticate.

Besides RFC 4164, the American institute for standards and technology (NIST) has also specified a good practice regarding key lengths and their validity periods. “The KSK needs to be rolled over less frequently than the ZSK. The recommended rollover frequency for the KSK is once every 1-2 years, whereas the ZSK should be rolled over every month for operational consistency but may be used longer if necessary for stability.” To NIST’s so called ‘special publications’ SP800-81r1 [25].

3.4.4 Traversal

The previous sections gave an introduction to what DNSSEC offers. Many details of DNSSEC have been discussed, however not all knowledge about DNSSEC has been brought together yet. That is done in this section, by showing a DNSSEC traversal – comparable to what is done for the regular DNS in section 2.3.3.

It is however good to explain some extra details, that until now have not been treated. DNSSEC is – as the name suggests – an extension to the regular DNS. It also uses port 53, and both UDP and TCP as its transport protocols. Also the packet format has not changed. Nevertheless, DNSSEC introduced some new resource record types and – as will be shown in the

[4]Note that the term Key Signing Key (and also the later to be introduced Zone Signing Key) is not well chosen, in the sense that it both are keypairs. So whenever a KSK signs something, this is signed by using the KSK’s private part. Validation takes place by using the public part of a KSK or ZSK respectively.

topology section – two new header bits. There is however no way for a client to indicate to an authoritative name server it wants to receive DNSSEC responses. DNSSEC solves this by adding a special flag, that is part of EDNS. EDNS (or: EDNS0) is an extension mechanism that is proposed [60] to be able to include more flags in a regular DNS packet. This is done by attaching so called OPT-type records to a packet's additional section. Besides, EDNS also makes the DNS capable of handling packets larger than 512 bytes, when being used over UDP.

A client can indicate it wants to receive DNSSEC responses, by using the so called "'DNSSEC OK' (`DO`) bit" [26] in the EDNS' OPT-type record. If a server supports DNSSEC, it leaves the `DO` flag set. Otherwise it clears the flag, something that automatically happens if an authoritative name server does not support EDNS. Because it will not include an OPT record in the response, since it implicitly thinks query packets do not contain resource records.

Let's return to the DNSSEC traversal. Before explaining this traversal, some assumptions have been made: It is assumed the root name servers are fully DNSSEC ready, and the root zones have been signed. This also means DNSSEC compatible clients have the public keys of the root zone securely retrieved and available. How key material should be published is described in [11]. Validation after retrieving trusted keys is required, and done by using mechanisms that are out of scope of this thesis, such as PGP. At this moment, no system has been developed that automates the process of obtaining the root's keys and validating them. Therefore, at this point a data manager has to manually set a trusted key.

It should also be remarked this traversal is purely educational, therefore quite some details have been simplified[5].

Just like in the regular DNS traversal example, a client wants to resolve `foo.example.org`, only now securely by using DNSSEC. Figure 3.4 illustrates the entire DNSSEC traversal, notice the yellow circles that point out the order in which a query or response is made. In the list below these circles will be discussed respectively.

0. This is the starting knowledge that every DNSSEC capable client knows by heart: the already trusted keys. It is needed to tackle the chicken and egg problem – already discussed during a regular DNS traversal, although now – for key material. As of July 15, 2010 these trusted keys should be known by DNSSEC aware hosts around the globe, because at that time the root zones have been signed and key material has been published.

[5]Examples hereof: DNSKEY-type resource records are never attached in responses when they are not explicitly queried. In the traversal only in the lowest level zone a distinction between KSKs and ZSKs has been made. Regularly every zone makes this distinction, due to the benefits explained in the previous section.

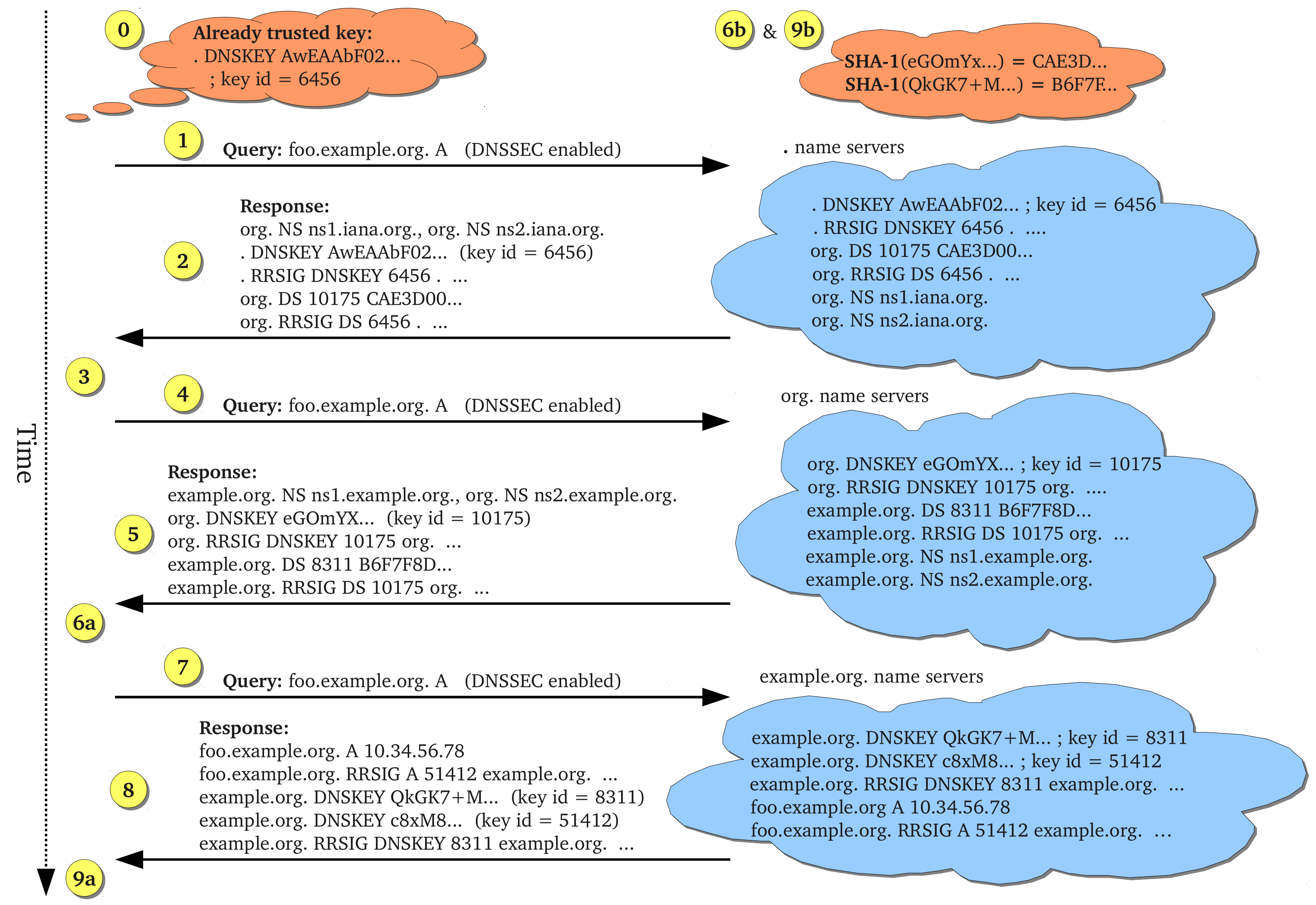
Time
0
Already trusted key:
. DNSKEY AwEAAbF02...
; key id = 6456
6b & 9b
SHA-1(eGOmYx...) = CAE3D...
SHA-1(QkGK7+M...) = B6F7E...
1
Query: foo.example.org. A (DNSSEC enabled)
. name servers
. DNSKEY AwEAAbF02... ; key id = 6456
. RRSIG DNSKEY 6456 . ....
org. DS 10175 CAE3D00...
org. RRSIG DS 6456 . ...
org. NS ns1.iana.org.
org. NS ns2.iana.org.
2
Response:
org. NS ns1.iana.org., org. NS ns2.iana.org.
. DNSKEY AwEAAbF02... (key id = 6456)
. RRSIG DNSKEY 6456 . ...
org. DS 10175 CAE3D00...
org. RRSIG DS 6456 . ...
3
4
Query: foo.example.org. A (DNSSEC enabled)
org. name servers
org. DNSKEY eGOmYX... ; key id = 10175
org. RRSIG DNSKEY 10175 org. ....
example.org. DS 8311 B6F7F8D...
example.org. RRSIG DS 10175 org. ...
example.org. NS ns1.example.org.
example.org. NS ns2.example.org.
5
Response:
example.org. NS ns1.example.org., org. NS ns2.example.org.
org. DNSKEY eGOmYX... (key id = 10175)
org. RRSIG DNSKEY 10175 org. ...
example.org. DS 8311 B6F7F8D...
example.org. RRSIG DS 10175 org. ...
6a
7
Query: foo.example.org. A (DNSSEC enabled)
example.org. name servers
example.org. DNSKEY QkGK7+M... ; key id = 8311
example.org. DNSKEY c8xM8... ; key id = 51412
example.org. RRSIG DNSKEY 8311 example.org. ...
foo.example.org A 10.34.56.78
foo.example.org. RRSIG A 51412 example.org. ...
8
Response:
foo.example.org. A 10.34.56.78
foo.example.org. RRSIG A 51412 example.org. ...
example.org. DNSKEY QkGK7+M... (key id = 8311)
example.org. DNSKEY c8xM8... (key id = 51412)
example.org. RRSIG DNSKEY 8311 example.org. ...
9a

Figure 3.4: *Graphical representation of a DNSSEC traversal*

This knowledge can be compared with the knowledge of the root server IP addresses, like the regular DNS (and DNSSEC too) requires. Notice the public key belongs to the root (notice the `.` up front), it is represented using an encoding that makes it able to express binary in ASCII-characters, more on this later. The key id (or: key tag) can be generated by using a standardized algorithm and the specific key as input. Remark however a key id is not uniquely identifying, but it gives a good indication to select the right key when a finite set of keys is present.

It should also be noticed that other points in the DNS tree can be taken as a trusted starting point. This was a popular phenomenon when for example the `.se` zone was signed and the DNS root was not. A DNSSEC client could then add the public keys of the `.se` zone to its trusted keys, to be able to verify every DNSSEC capable zone in the `.se` domain.

1. Like in the regular DNS, a DNSSEC aware client also has a list of root server IP addresses available. So an A-type query is sent towards one of the root servers, although now it indicates it would like to receive DNSSEC responses. This is done by setting the DNSSEC okay (`DO`)-bit.

2. In the response of the root server, several resource records can be found. Just like in the standard DNS, there are NS-type responses, that refer to the `.org` name servers.

   Next, a Delegation Signer record can be distinguished. This record is accompanied by a signature, in the form of a RRSIG-type resource record. By looking at the root (`.`) RRSIG-type record, it is known this RRSIG record contains a signature that is generated over the DS-type RRset, by using a key with id 6456. This key can be found in the root (`.`) zone.

   The DS-type record states the root server delegates trust for the `.org` zone towards a key record somewhere down in the hierarchy with a key id equal to 10175, and a digest starting with `CAE3D00`.

   Finally, a DNSKEY-type record of the root is returned, together with a corresponding signature. This is indeed the key that signed the DS-type record, but it is also used to authenticate itself, i.e. the DNSKEY-type record.

   Notice there is no signature included for the NS-type records. This might seem awkward, but it protects integrity: because the root servers are no authority for the `.org` zone. So a DNSSEC aware name server does not sign data it is not authoritative for.

3. The first verification step can now start. A DNSKEY-type record for the root was received. Since there already is a trusted key for this

zone, both keys are compared. It can be seen the received key matches the already trusted key (shown in item 0). The signature for the DNSKEY-type record can now be verified, claiming authenticity of the key – indeed, this is a bit superfluous since authenticity is already proven by the trusted key comparison.

Finally, the signature of the delegation signer record is verified. This signature is verified by using the key with id 6456, that is the already trusted root zone key. Furthermore, the client will remember that trust for the `.org` zone is delegated towards a key with an id equal to 10175, and the associated cryptographic hash.

4. If everything verified okay, one of the `.org` name servers – as listed in the previous response – will be contacted. The same query as in item 1 will be used.

5. The response that is received is similar to the one of item 2. Again, a signer delegation to a zone below is given (trust for `example.org` is delegated to a key with id 8311 and corresponding digest). And this signer delegation is signed by using the zone's own key (with id 10175), that is also delivered with this response, including a signature for the key.

6a. Now a more interesting verification step can start. It was noted that trust for the `.org` zone was delegated to a key with id 10175, and a digest starting with `CAE3D00`. This matches the id of the key that has just been retrieved. According to circle 6b, also the digests turn out to be equal. This means that authenticity of the zone can be guaranteed by using this key to verify all other RRsets.

The signature that accompanied the key will now be verified, by using the key that was just authenticated. And finally the client will note a delegation signer is found. This time trust for the `example.org` zone is delegated to a key with id 8311, and a digest starting with `B6F7F8D`. Of course, the signature of this DS-type record will be verified too with this zone's key.

7. If all signatures validated okay, a new query towards to the `example.org` name servers will be made. Again, this is just the same query as the ones directed to the root-, and `.org` name servers.

8. This is the most interesting response in a DNSSEC traversal. Instead of a referral to another name server, the actual result is returned, also accompanied by a RRSIG-type record.

There are also two DNSKEY-type records included in the response, instead of only one, as seen before. One having key id 8311, the other with key tag 51412. Although not explicitly shown in the illustration, key 8311 has a larger key size of 2048-bits, compared with a key size of 768-bits for key 51412. Indeed, key 8311 can be considered the Key Signing Key (KSK) of the `example.org` zone, while key 51412 is the Zone Signing Key (ZSK).

9a. The client remembered from the `.org` zone response, it should trust a key with id 8311 and a corresponding digest in the `example.org` zone. Circle 9b shows that the received DNSKEY-type record with tag 8311 fulfills the digest requirement, this means this key is authentic. Now look at the signature of the A-type record. This signature is generated with a different key (with tag 51412), that is not yet authenticated. This indicates there is a distinction between KSKs and ZSKs.

    So first this 51412 ZSK will be authenticated. This is done by checking the signature for the DNSKEY-type record that is signed with the already authenticated 8311 KSK. If this turns out okay, the ZSK with key tag 51412 can be considered authentic too. This implies the actual result's (the A-type record) signature can be verified, using the 51412 ZSK key.

Notice that during this traversal again everything went okay, i.e. all name servers responded, every query was positively answered, always DNSSEC responses were given, and all signatures turned out okay. However, for each problem that might occur scenarios have been specified how to deal with this. For the simplicity of this text however, all these details were intentionally left out.

3.4.5 Topology

Now that it is known how a DNSSEC traversal in general works, it is good to also look at how a DNSSEC topology would look like. In comparison with the topology that is known from the regular DNS chapter.

Considering the global topology map (as shown in Figure 2.4), not much is changed in the DNSSEC environment. There still is a separation between authoritative name servers, and caching (or: resolving) name servers. Also stub resolvers still do exist.

The difference between a regular DNS and a DNSSEC topology exists in the question where the verification takes place. To be able to communicate towards a host, a client would like to receive either validated, or non-validated responses, two new header options have been introduced by DNSSEC. It is good to explain both first.

During the explanation of the regular DNS header in section 2.3.5, three so called `Z`-bits were discussed. Although at that point, it was told these bits were regularly 0, but in the future they might be allocated for new features. DNSSEC used this possibility.

The last two bits of the Z-bits found a new destination. The first is the AD-bit, where AD stands for 'authenticated data'. The second bit is the CD-bit, that stands for 'checking disabled'.

The AD-bit only plays a role in response DNS packets, and indicates to a client that the bit is only set when "the resolver (...) considers all RRsets in the answer section and any relevant negative response RRs in the authority section to be authentic" [14]. Regular authoritative name servers will never send responses with this bit set: because they never verify signatures that they include in responses. This means this bit will only be touched by name servers that actually have checked signatures, such as resolving name servers.

The CD-bit will be set by a client (thus in a query) to indicate the client does not want the name server to verify signatures. Meaning the name server should simply pass the response along, even when signatures turn out to validate wrong. Both header bits are used to let regular DNS topology also work in a DNSSEC matter.

Let's look back at the regular DNS topology – as portrayed in Figure 2.4 – and focus on all different parts of the illustration below.

Authoritative Name Servers

If the authoritative name server supports DNSSEC, it should implicitly support EDNS. This has two major reasons. First is to have support to receive and understand the DO-bit, that indicates a client is willing to receive DNSSEC responses. Another reason is that EDNS gives the ability to support larger DNS packets, when being used over UDP. Even 65,535 bytes big packets could now be sent using a single UDP packet. This would prevent clients switching constantly to TCP, when the old 512 byte barrier would be reached. Because DNSSEC responses are usually several factors larger than regular DNS responses, mainly due to the addition of signatures and key material.

Next, a DNSSEC capable authoritative name server should of course be able to generate signatures for all authoritative data is serves. So far, it was unclear whether this signing process would be done in a realtime (for every query a 'new' signature would be generated), or offline fashion (every now and then a 'new' signature will be generated). Doing this in realtime did not seem feasible due to speed limitations and concerns regarding stability and availability. Therefore, signatures are generated in an offline fashion. Meaning that from regular zone data once in a while a signature is being generated, independent of the serving process. Resigning usually takes place when zone data has been altered. Notice that in an RRSIG-type record it can be seen when the record has been signed, and till when this signature can be considered valid (respectively the record's `signature inception` and `signature expiration` fields).

Furthermore the authoritative name server should be capable of sending out the new DNSSEC resource record types (DNSKEY, RRSIG, DS, and NSEC and/or NSEC3). Remark that all these records have been generated before they are served.

Authoritative name servers will never respond with the `AD`-bit set. Because they only deliver the records accompanied with signatures, and do not do any form of validating. If a client would send a query with the `CD`-bit set, an authoritative name server would not do anything with it, and clear the bit in any response.

Caching Name Servers

Caching name servers have – as discussed earlier – in fact two sides. One side receives all query requests from clients, and eventually respond the result back to the client through this channel – it can be considered a server in this way. While the other side is in charge of resolving the received queries, by sending queries to authoritative name servers around the globe. So a caching name server is on the one hand a server, and on the other hand a client. In the middle of these two sides, the cache resides. RFC 4035 [14] makes a distinction between the two. It calls the server side the 'recursive name server', and the resolving client side the 'resolver'.

A DNSSEC capable caching name server will try to send out queries indicating it is able to handle DNSSEC responses. This means it understands the newly introduced DNSSEC record types, and also how to deal with them when they are included in a response.

Regularly all caching name servers will validate DNSSEC responses on their own. This means that when a DNSSEC response is received (on the resolver side), the signatures will be checked. It depends on the caching name server's policy what to do when a record's signature turns out to be rogue. Such policy is important in DNSSEC and expresses crucial decisions, i.e. should a rogue response be forwarded towards the client, and may it be cached?

When a client at the recursive name server side did not request any DNSSEC support, i.e. it did not send the `DO`-bit along in the query, it will receive a regular response. Just like a caching name server would react when it was used in the standard DNS setting.

However, when a client does request DNSSEC support, a caching name server will take care of the `AD`-bit in the response to the client. The `AD`-bit will only be set when the caching name server "considers all RRsets in the answer and authority sections of the response to be authentic", or when it "considers all RRsets in the answer section and any relevant negative response RRs in the authority section to be authentic" (both taken from [14]). When one of these guarantees cannot be met, the AD-bit will not be set.

If a client sets the 'checking disabled' (CD)-bit, a caching name server will not do any validating of signatures. It will simply act as a proxy, and pass along all the raw data it received including any of the DNSSEC resource records. This data might even come from the cache, and was flagged rogue because it contained an invalid signature. In this case all responsibility lies within the client. So setting the CD-bit in a query, simply states the client will do the verifying of signatures. The caching name server will respond to this query with the CD-bit still set. Notice however, the CD-bit does not influence the behavior of the AD-bit. A caching name server might set the AD-bit in a response, when the CD-bit is set in the corresponding query, indicating that the data is authentic, according to the caching name server's policy.

Stub Resolvers

Stub resolvers are usually the entities to speak to when an actual client (such as programs) wants to resolve a domain name on a local system. These resolvers are generally integrated in the operating system and do not involve much code, as they depend on one or more caching name servers.

If a stub resolver would not be DNSSEC capable, a caching name server will not receive a DO-bit in the query. So it will respond as it would regularly do, without any signatures attached to the response. Notice that a stub resolver is in this way still vulnerable to any active attack. Although, it could use DNSSEC either way on its resolver side, to authenticate the responses it receives on that side.

If a stub resolver is DNSSEC aware, there are still two options. It will either do verification on its own, or it will rely on the caching name server to do this for him. In both ways, the stub resolver will send its queries containing the DO-bit.

Doing verification on its own, might not seem obvious. But imagine some client wants to deploy its own policy regarding DNSSEC, instead of following the policy the caching name server operates on. Or what if a stub resolver does not actually trust the caching name server.

To indicate to the resolving name server, it would like to retrieve all records – biased or not – the stub resolver should set the CD-bit in the query. The caching name server will now respond raw data, including signatures and such. Notice however, that data that resides from the cache, might be double verified. Once by the caching name server before it entered the cache, and once by the stub resolver.

If a stub resolver is DNSSEC capable, but leaves signature verification to the caching name server, it must fully trust the caching name server. Plus implicitly its policy. RFC 4035 states that "responses received by a (...) resolver are heavily dependent on the local policy of the (...) recursive name server (...). Therefore, there may be little practical value in checking the status of the `AD`-bit". Remark this brings new security issues, that will be discussed in this chapter's security section.

Actual Clients

Actual clients do not consider any DNSSEC options. Usually they just want to resolve a certain domain name. However, for some services authenticated DNS responses are very valuable. Examples of these services are mail daemons (sending email to the right server, instead of to the attacker's one), and web browsers (directing a person to the Internet banking's website, and not to an adversary's copy of it). There is however nothing specified yet, how an actual client can communicate this to the operating system's stub resolver. It should therefore trust the system's administrator it configured the stub resolver to either use a DNSSEC aware (and enabled) resolving name server, or to verify signatures on its own.

3.4.6 Data Managers

The common operation of DNSSEC is now clear. However, quite some things have changed for data managers (i.e. persons responsible for data that is being served by authoritative name servers) in comparison with the regular DNS.

The initialization process for a data manager to become DNSSEC capable, starts by generating a key signing keypair. This keypair should be relatively large, since it will be used for a long period. Furthermore a digest of this key should be transmitted to the data manager that is responsible for the zone one level up in the hierarchy. The DNSSEC RFCs do not specify how this exactly should happen. It is however obvious this should be done in a secure matter.

Next, a zone signing keypair should be generated. For its authentication it should be signed with the previously generated KSK. Now that there is a ZSK, the entire zone can be signed by using this key. Although first, either the NSEC- or NSEC3-type records should be generated. Now the RRSIG-type records can be generated.

All these actions concern the initiation of DNSSEC capability. Although usually DNS data is about to change. In the regular DNS this was already the case and this will still happen in a DNSSEC environment. Although this time, the signature for a record that has been added or modified, should be regenerated. In case of a record response name modification or deletion, also a different or new NSEC- or NSEC3-type record should be created respectively. Such denial of existence record should also be accompanied by a signature.

Besides resigning when the plaintext data has been altered, signatures should also be recreated when their lifetime is about to expire. During discussion of the RRSIG-type records, a creation and expiration date were distinguished. Notice these times were absolute, instead of relative, as all times in the DNS were until now. This implies that every party involved in the DNSSEC environment should have a reliable time source. This could be accomplished by using a synchronized network time protocol (NTP).

Then there is the earlier mentioned key replacement (or: key rollover). Keys do not have an explicit expiration date, they should however be replaced once in a while – depending on the algorithm and key size – before an attacker can deduct any information from it. When a zone key is rolled over, the entire zone has to be resigned, including the signature for the DNSKEY-type RRset. On the other hand, when a key signing key is rolled over, only the RRSIG record of the DNSKEY RRset should be resigned by both the ZSK and new KSK. Plus, the data manager of the upper level should be notified that a new KSK is in place in one of its child zones. The digest of this new KSK should be put in the zone's DS-type record in the upper level zone.

Computational power is needed to generate cryptographic signatures and hashes. When the `.nl` zone started experimenting [35] with DNSSEC around 2003, its non-DNSSEC zone file was around 40MB in size. It took 90 minutes to generate a DNSSEC enabled zone. It is not mentioned what hardware was used.

Besides computational power, generating DNSSEC capable zone data also requires more storage, due to the addition of RRSIG-, DS-, and NSEC- or NSEC3-type records. The zone size will increase with a factor between 4 and 12 [5], depending on the chosen digest and algorithm, key size, plain record lengths, number of RR types per response name, and what denial of existence RR type is chosen. The earlier mentioned `.nl` zone file increased to 350MB in size. The addition of denial of existence records are the major reason for this increase, because for every unique response name such record must be included.

Due to this increase a server's memory load will also grow. In a measurement, conducted by RIPE NCC [39] for the K-root server and RIPE's own primary name server, it is shown memory usage will increase "with less than 5%". While CPU usage of an authoritative name server (notice this does not include the signing process) will grow with "4 to 5%". Next, also bandwidth usage will expand. The measurement shows this bandwidth increase is around 10%. In this paper, both the BIND name server and the NSD name server were used.

Also NIST performed benchmarks regarding DNSSEC performance [5]. These numbers conflict on some parts with the conclusion of the RIPE paper. For example the memory usage measurement heavily varies. In the

NIST measurement zone files have been used, that varied between 500 and 30,000 response names. Every of such response name had five different RR types. When serving these zones, memory usage increases between 9 and 209% – when a 1024-bit key was used. Numbers get even higher when a 2048-bit zone signing key was used.

Also the bandwidth increase numbers vary. The NIST measurement distinguishes four different traffic traces, that consist of real but anonymized data. The difference in bandwidth usage, between a non-DNSSEC traffic trace and one in which 90% of the queries is using DNSSEC, lead to an average increase of 437%. NIST did not measure CPU load while serving data, it did however look at the startup (or: load) time of the authoritative name server. So no comparison can be made regarding that measurement.

Doing all the needed transitions for DNSSEC deployment by hand is not done easily, even for a small number of zones. Therefore several toolsets have been developed to make life easier for data managers. OpenDNSSEC [10] is one of them, and states that it "takes in unsigned zones, adds the signatures and other records for DNSSEC and passes it on to the authoritative name servers for that zone". This simplifies DNSSEC deployment and maintenance considerably, even for non-DNSSEC aware data managers.

3.5 Security

In the regular DNS chapter's security section, many vulnerabilities of the DNS protocol have been discussed. This section will discuss each of the attacks, however now in the DNSSEC environment. Furthermore, some new attacks will be introduced.

3.5.1 Passive Attacks

As mentioned in the regular DNS chapter, passive attacks are attacks that do not influence the regular operation of a protocol. This is generally done by eavesdropping (or: sniffing) on a communication line at which the protocol operates.

In the regular DNS all queries and responses are sent in plaintext over the line. DNSSEC still does the same. An attacker is therefore still able to see what hosts on the network are doing on other networks (such as the Internet). This because every website visit is preceded by a DNS or DNSSEC query.

Detailed traffic analysis might even answer questions such as what websites are popular by hosts on a network, or what electronic banking website is used.

3.5.2 Active Attacks

Active attacks are much less effective in DNSSEC than in the regular DNS. When an attacker was able to actively interfere with the communication in the DNS, it in fact could control the entire DNS. One of the main objectives of DNSSEC was to supply integrity for the DNS, making such attacks not exploitable anymore.

This is due to the authenticity DNSSEC provides. Imagine a DNSSEC enabled query (the `DO`-bit is set) is send to resolve `www.example.org`. An attacker would forge, and generate an artificial response including the correct query name, identifier, type, and class and destined to the right source port. (Notice this is easy to find out if the attacker is in full control of the communication line.) Only this time, the response includes an address owned by the attacker. The client will now be able to detect these modifications, because the RRSIG-type record that accompanies the A-type record will not verify correctly. At least, not with the valid zone signing key.

Even if the attacker influences all the queries earlier in a traversal. For example by removing the right DNSKEY-, DS-, and RRSIG-type records, and including corresponding ones created by the attacker. In this way, even a rogue zone signing key could be included. Nevertheless, an attacker will not be able to successfully fulfill this. Since every client needs to have at least one trusted key that starts the authenticated chain – usually the root of the DNS structure.

DNSSEC does not sign data it is not authoritative for. This means for example that NS-type records will never get signed, since they are not in the current zone's authority. An attacker might think this opens new attack vectors. For example by injecting false responses from the `.org` name servers.

Assume the `example.org` domain is DNSSEC capable, the `.org` zone should therefore have a NS-type delegation to the `example.org` name servers, notice that this RRset does indeed not include a signature. Together with this delegation, also a DS-type record for the `example.org` can be found in the `.org` zone. If an attacker would modify the NS-type records and leave out the DS-type records, plus its signature, a client might think the zone lives at another name server, and is not DNSSEC capable. This sounds like a working scenario. However since no DS-type record is included, the `.org` name servers should include a denial of existence record stating there is no DS-type record for the `example.org` domain. An attacker is unable to forge this.

DNSSEC is not able to protect withholding of DNS queries or responses by an adversary. If an adversary would for example block a DNSSEC query by a client, the client will never receive a response, thinking the server is unavailable at that time. Such attacks lead to availability issues.

On the other hand, DNSSEC does protect against random bit flipping of DNS responses, because it supplies signatures for these responses. Bit flipping of either the data itself or of the included signature would lead to unverifiable resource records, that will eventually be denied. Notice however, that DNSSEC queries are still subject to bit flipping, since no integrity constraint is applied to queries. Meaning that queries can still be altered, without the receiver knowing.

In fact, an attacker could for example remove the EDNS section of a query, making it a regular one instead of a DNSSEC query. Nevertheless, if the client would expect a DNSSEC response – because of a DS-referral earlier in the traversal – it will not except a non-DNSSEC response.

Keep however in mind that for non-DNSSEC queries no protection can be guaranteed. Everything that is noted in the regular DNS security section on active attacks, still applies to queries without the `DO`-bit set.

3.5.3 Passive Cache Poisoning

Although the passive cache poisoning attack was already rendered harmless in the regular DNS due to in-bailiwick checking, it is good to see whether DNSSEC adds protection for such attacks.

This attack added rogue glue records to a response. So if a client resolved the A-type RR from the domain `www.example.org`, it received a correct response from the authoritative name server. However, in the additional section of the answer a resource record was included that stated `www.google.com` lived at an attacker owned address. As mentioned, the in-bailiwick check would have ignored this glue record, because the name server for the `example.org` domain is no authority for the `.com` domain (not even for the `google.com` domain).

DNSSEC does not give any extra protection against this attack, because it does not supply signatures for data it is not authoritative for. This means that all RRsets in a response that are not in-bailiwick, will not get a corresponding RRSIG-type record. Therefore authenticity of these records cannot be guaranteed. Still, due to the in-bailiwick protection available in all current resolver implementations, this will not have any exploitable effect.

3.5.4 Identifier Guessing and Query Prediction

Identifier guessing and query prediction, was the first attack that actively interfered the DNS protocol, without being in full control of the communication line. This was done by letting a victim visit an attacker's website, to retrieve (non-existing) images from the target domain. The attacker now knows the victim's IP address (due to the visit of his website), and the query name, class, and type (due to the image retrieval of the target domain). The adversary will now send many DNS response packets, containing the query name, class, and type of the target domain. The answer section will contain a resource record that would point the target domain to an address the attacker choses.

The likelihood of a successful attack is not very high, if the resolver of the victim uses a cryptographically random source for both its DNS packet identifier and source port. However, many implementation lacked this requirement, making it easier for an attacker to successfully deploy this attack.

For this attack the same argumentation regarding DNSSEC goes up as used in the active attack subsection. If a client sent a DNSSEC enabled query, and the attacker would be able to correctly guess all fields, it can insert a rogue answer. Although now the client will be able to detect this, since no signatures are included, or – if included – they turn out to validate wrong.

It might seem easy for an attacker to avoid all DNSSEC signature verification, by removing all DNSSEC information (such as DNSSEC specific resource record types, the `DO`-bit, and clearing both the `AD`- and `CD`-bit) from a valid response. But a resolver knows whether it should expect a DNSSEC response, or not. This is caused by either two factors: in an upper level zone trust was referenced to the zone that is now being queries (by using a DS-type resource record), or because trust in the zone is entangled by an already trusted key. So if a DNSSEC aware resolver receives a non-DNSSEC response, after sending a DNSSEC enabled query it will regularly – depending on the policy – not accept regular DNS responses.

An attack that can still be applied is described hereafter. Assume the `.org` zone is DNSSEC capable. A client will ask for `www.example.org` to one of the `.org` authoritative name servers. It will also indicate it is DNSSEC capable (i.e. the `DO`-bit will be set). The `example.org` zone itself is *not* DNSSEC capable. A valid DNSSEC response will look like the one that is portrayed in Figure 3.5.

```
;; QUESTION SECTION:
;www.example.org.       IN  A

;; ANSWER SECTION

;; AUTHORITY SECTION:
example.org.     86400  IN  NS    ns1.example.org.
example.org.     86400  IN  NS    ns2.example.net.
example.org.     7200   IN  NSEC  org. NS RRSIG NSEC
example.org.     7200   IN  RRSIG NSEC 5 2 7200 201007..
                                     201007.. 6870 org. OcHKzY6...

;; ADDITIONAL SECTION:
ns1.example.org. 86400  IN  A     192.168.86.113
```

Figure 3.5: *Valid DNSSEC response for a non-DNSSEC enabled child zone*

Of course, the `.org` name servers do not have an answer for `www.example.org`, it does however indicate that `ns1.example.org` or the `ns2.example.net` name server know more. A NSEC-type record is included, to show that the `example.org` zone is not DNSSEC capable. Since it shows there is no DS-type record in between the 'gap', only NS-, RRSIG-, and NSEC-type RRsets exist. To authenticate this denial of existence, an RRSIG-type record has been included. In the additional section the interesting part of the attack takes place.

In the additional section one glue record can be found. It states that `ns1.example.org` lives at `192.168.86.113`. It might seem strange there is no resource record for `ns2.example.net`. This is however intentionally left out, since the `.org` name servers are no authority for the `example.net` zone. The attack now is obvious: the address of `ns1.example.org` can be altered, without the recipient knowing. In this case theoretically 50% of the queries will be answered by an attacker's chosen name server, if the injection of this packet was successful.

This attack is caused by the decision of DNSSEC not to sign non-authoritative records, and implies that even if a zone is DNSSEC capable, its direct children zones that are not DNSSEC capable will not be protected. Remark however this attack would not work if: the name servers of the domain were all out-of-bailiwick (no glue records to modify), the `example.org` zone itself was DNSSEC capable (DS-type record included, therefore in the next iteration signatures would not verify okay), or when some of the packet identifiers were not guessed correctly. It must be clear that before a successful injection the rogue packet still needs to win the race with the valid response. Another disadvantage is that if an injection was not successful, the attacker should wait for the additional record's TTL, before it can try again.

Again remark that non-DNSSEC queries are not covered by the DNSSEC guarantee. So regular DNS queries are still vulnerable to this attack.

3.5.5 Active Cache Poisoning

Active cache poisoning (also known as the Kaminsky attack [27]) combines both earlier discussed attacks: identifier guessing plus query prediction, together with the addition of rogue glue records. Although these rogue glue records are included in such a way they circumvent the in-bailiwick checks, simply by being in the domain's bailiwick. An example of a by the attacker generated response in the regular DNS can be seen in Figure 2.9. Notice that for this attack the client again has to be flooded with crafted DNS responses, to increase the success rate.

Figure 3.6 illustrates what a valid DNSSEC response for such type of attack should look like. Remark this response comes from the name server that serves the `example.org` domain. Notice the target is the `example.org` zone, that is served by the two attacker owned servers running at 66.66.66.66 and 66.66.66.67. The domain contains only one record: www.example.org.

```
;; QUESTION SECTION:
;001.example.org.          IN  A

;; ANSWER SECTION:

;; AUTHORITY SECTION:
example.org.      7200     IN  NSEC  www.example.org. NS RRSIG NSEC
example.org.      7200     IN  RRSIG NSEC 5 2 7200 201006..
                                201006.. 1337 example.org. aKx6WGiS...
example.org.      172800   IN  NS    ns1.example.org.
example.org.      172800   IN  NS    ns2.example.org.
example.org.      172800   IN  RRSIG NS 5 2 172800 201006..
                                201006.. 1337 example.org. n6lWOPcH...

;; ADDITIONAL SECTION:
ns1.example.org.  2592000  IN  A     66.66.66.66
ns1.example.org.  2592000  IN  RRSIG A 5 3 2592000 201006..
                                201006.. 1337 example.org. K6wIPfVk...
ns2.example.org.  2592000  IN  A     66.66.66.67
ns2.example.org.  2592000  IN  RRSIG A 5 3 2592000 201006..
                                201006.. 1337 example.org. 38byWLkB...
```

Figure 3.6: *Valid DNSSEC response to actively poisoning a resolver*

There is again no actual answer for the 001.example.org question. Let's check the authority section. Because a denial of existence response should also be authenticated, a NSEC-type record is included. It states that between example.org and www.example.org there are no other records. Since example.org is one level up in the hierarchy, it means www.example.org is the first record in the example.org zone. There is also a signature provided for the NSEC-type record. The RRSIG-type record indicates it used the RSA/SHA-1 combination to create the signature. The signature inception and expiration dates are shortened for simplification. Key 1337 from the example.org zone was used to generate the signature (algorithm #5 in Table 3.1). The signature itself is also shortened for simplification.

The NS-type RRset in the authority section has its own signature that is signed with key 1337 from the example.org zone. The authority section can be compared with the response that was given in the regular DNS, only now accompanied with signatures for the existing RRsets.

Now look at the additional section. This is where previously the tricky part of the attack took place. Because both response names are in-bailiwick with the example.org domain, both answers will be accepted. DNSSEC aware authoritative name servers do not supply signatures for data they are not authoritative for. The nice design of the Kaminsky attack, to be in-bailiwick with the crafted additional section response, will now be a disadvantage. Because all records that are in the domain's authority should be accompanied with a signature. That is what can be seen in the figure, both records have their own respective RRSIG-type record.

A DNSSEC capable client will now verify both the supplied signatures. Making it unfeasible for an attacker to circumvent these checks, even when no signatures or false ones are provisioned. Since an attacker must be able to control the DNSKEY-type record of key 1337 in the `example.org` zone. This would not be possible due to the authenticated chain that is created with delegation of trust through the DS-type resource records higher in the hierarchy. Assuming the root is already signed and known to the client, or that the client has a trusted starting point somewhere under the root.

3.5.6 Amplification Attack

In short, amplification attacks involve the usage of the DNS protocol to attack other hosts' availability. This is done by spoofing the client's IP address, which is possible due to the usage of the stateless UDP protocol.

In the regular DNS chapter, the term 'amplification factor' was introduced. This factor indicates how many times a data stream can be amplified, to attack another host. If for example a 33-byte DNS query is sent to some authoritative name server, a response of 161 bytes could be returned to a spoofed IP address. This is an amplification factor of 4.87.

The introduction of DNSSEC brings new vectors in this kind of attack. Because new resource record types have been introduced, that in general have a much larger size than earlier existing RR types. Take for example the DNSKEY-type records. These records store key material depending on the algorithm that is used. Subsection 3.4.2 showed the RSA/SHA-1 combination is mainly used. It does not give an indication what key sizes are generally used, but assume this to be 1024-bits. This would mean the `public key` field of a DNSKEY-type record itself, would already need 132 bytes. Assuming a zone consists of more than one key (due to the separation of key- and zone signing keys), even larger amplification factors can be achieved.

Let's look at a real world example. The query asks for DNSKEY-type records, of the `.br` zone (that is already DNSSEC capable) to one of the Brazilian country code authoritative name servers (`a.dns.br`). This query is in total 31 bytes long, the response that is received is 945 bytes in size. Resulting in an amplification factor of 30.48.

The name server responded three DNSKEY-type records, two with a 1280-bits key size, and one with a 1152-bits key size. The length implicitly indicates what keys are KSKs, and what is the ZSK. Together with these three records, two RRSIG-records are included. Both records supply signatures for the DNSKEY-type records.Each signature by itself is 160 bytes long.

Summing everything up, including the length of all mandatory fields (response name, class, type, TTL, etcetera), 945 bytes can be obtained.

The DNSCurve website [6] found even some larger amplification factors. The largest factor currently known regarding DNSSEC enabled name servers is to query for `ANY` record from the `.se` zone, to one of the Swedish country code name servers (such as `a.ns.se`). This 31 byte query packet, results in a 3997 byte response. This response packet includes 23 regular answers, and 22 additional answers. Causing an amplification factor of 128.94.

This means DNSSEC opens a new world for amplification attacks. A client with a 1Mbit/s connection would now be able to generate a UDP flood of 128.94Mbit/s. This will most definitely get the target host down. Things could get even worse if this attack is done in a coordinated distributed approach. Remark there are also seven authoritative name servers for the Swedish ccTLD, making the attack a bit harder to detect by traffic monitoring.

Notice that if different source IP addresses are used that originate from the same subnet (for example `10.86.11.3` and `10.86.11.26`), the responses will normally take the same route towards the host. This would cause all communication nodes in between, to handle a huge amount of data. Such silting communication lines result in higher latency and maybe even malfunction.

3.5.7 Hierarchical Trust

Hierarchical trust on itself is not an attack. It was mainly a reminder that 'taking care of your own business' does not give a complete indication a host is secure in a DNS environment. If a host is compromised that serves DNS data high in the DNS hierarchy, this means that all zones underneath are vulnerable. In the sense they can be entirely left out, by modifying or removing the referrals (NS-type records) to these hosts.

In the DNSSEC environment this reminder still applies, but even stronger. Since DNSSEC introduces more 'connections' between zones. In the regular DNS situation, a connection between zones could only be made by referrals. DNSSEC introduced the DS-type resource record to be able to delegate trust of a zone lower in the hierarchy to a certain DNSKEY-type RR. This means another connection between zones.

If an attacker was able to compromise an authoritative name server in the regular DNS, it could redirect traffic to any referral. To do the same in the DNSSEC environment, the attacker should get hold of the private part of the keys used by the name server. Since no alteration can be done without regenerating the corresponding RRSIG-type records. These private parts should be securely stored. Sometimes this is done at so called Hardware Security Modules (HSM), or at smart cards. These solutions are however relatively expensive, making it harder for smaller organizations to use them. Therefore such organizations will store their private keys on local storage. Which would make it easier for an attacker to obtaining the private key material.

Notice however that most data managers have automated the process of regenerating signatures. Sometimes by making customers able to perform their DNS modification online by a webservice. In this way successfully attacking that webservice would be enough to compromise an authoritative name server's data.

Due to the earlier mentioned increase in connections, also another problem rises. How should a DNSKEY digest being transferred to an authority higher in the hierarchy. The DNSSEC RFCs do not mention how to do this – "this communication is an operational matter not covered by this document" [14]. Although it is clear this should be done in a secure matter, because trust for a zone depends on whether the right digest corresponding to the zone's KSK is put in the DS-type resource record. If an attacker would be able to interfere with this process, i.e. it can insert its own key's digest, it is again in control of the target domain. Nevertheless it should still actively attack the client, however, the packets should contain signatures originating from the inserted digest's key. Or, if the attacker would be able to also alter the name server addresses, the attacker is entirely in control.

Finally, it should be remarked that DNSSEC also introduces a new notion: authenticated chain of trust. This chain of trust roots at a certain point in the DNS tree and as of July 15, 2010 this can be the DNS root. The initial trust for the chain is enforced by supplying a trusted key. These trusted keys can be compared with the trust that is put in the root server list, that the regular DNS (and DNSSEC too) uses. Without this initial starting point of thed chain of trust, no guarantee of trust can be given in the entire DNSSEC architecture.

It should be clear by now that hierarchical trust is still of concern – and even more – in the DNSSEC environment.

3.5.8 Other Attacks

Other attacks are attacks that do affect the DNS (or DNSSEC) itself, but are not caused by the DNS.

The type of attacks discussed in the regular DNS chapter, can still be applied in the DNSSEC environment. Although it is good to point out the private key parts are of great value in the DNSSEC world. This means that special care is needed when dealing with this material. As mentioned before, hardware security modules could help in protecting these goods.

3.5.9 Timing Attacks

This subsection cannot be found in the regular DNS' security section. DNSSEC introduces the usage of absolute time in the DNS. The regular DNS only used relative times, that is why no section was dedicated to this kind of attacks.

Absolute time information is truly important in the DNSSEC matter and can be found in the RRSIG-type records. They state when the signature was created and when it will expire. Regularly the difference between the two is around a month. This means the signature can be considered valid one month after its inception. This validity period was built in to decrease the number of (offline) sign operations. If the signed RRset would not be altered in this one month period, no new signatures for the set would have needed to be generated.

There are also downsides on this operational benefit. One of which that is referred to as "DNSSEC suicide" [22]. When the validity period of a signature is about to expire, a new signature should be generated. If a data manager of a DNSSEC enabled domain forgets to do this (for whatever reason), affected resource record sets with such signature can be considered invalid, depending on the client's policy. This could make a website or service – that is addressed by this zone – unavailable. This phenomenon is called DNSSEC suicide.

Because of this attack, accurate time information is important when using DNSSEC. It was already made clear that this can be achieved by using a synchronized network time source (using the NTP protocol). But what if an attacker is able to skew the client's system clock. Either way, by putting the clock for- or backwards, this will influence the client's behavior when a DNSSEC signature will be verified. Therefore the system's clock should be accurate, and secure.

When time skips are performed at authoritative name servers, impact is of course much larger. Since all clients will retrieve signatures that are not created at the time that is mentioned by the signature. This could lead to unusable signatures where the validity period falls out of the real time's scope.

Remark that time alterations in the regular DNS do not have much influence. It would only lead to less effectiveness of the system's DNS cache, since TTLs are reached earlier (or later of course).

3.5.10 Replay Attacks

Replay attacks did not play a role in the regular DNS. However, not in the way it is represented here. Because the active attacks that try to win the race against valid packets in the standard DNS chapter, do in some way replay earlier sent or received packets. Nevertheless, in this subsection replay attacks are attacks that replay responses without altering them.

One attack that is described by [6] involves the exact replay of DNSSEC responses. Think of an organization that is about to move to another location. In the regular DNS the TTL will be decreased to a very low value, when this is about to happen. In this way, clients will not cache data received from the organization's authoritative name servers for a long period, making a move of services to other IP addresses easier.

In the DNSSEC environment things are a bit different. What if an organization moved over, and an attacker is able to buy the organization's old IP addresses. If the attacker saved all legitimate responses it received from the organization's DNSSEC capable authoritative name server in the days or weeks before the move, it is able to replay these responses originating from the acquired IP addresses. In this way it could intercept email traffic, or other data that was destined for the organization.

Although RRSIG-type resource records are accompanied by a validity period, these usually have a length of one month. Leaving the opportunity for an attacker to replay responses for less than a month.

A quick fix is of course to also lower the signature's validity period, as the move date is coming closer. This could however introduce usability problems, because data managers have to resign their data much more often.

The previous scenario might seem unlikely to happen, but there are other replay attacks that can constantly take place. Look back at the `.org` zone information that is portrayed in Figure 3.3. Now imagine all RRSIG-type resource records in that example have a validity period of one month. Assume the attacker saves a NSEC-type response, together with its corresponding signature. The NSEC record states there are no other domains between `alfa.org` and `charlie.org`.

Now the data manager adds the domain `bravo.org` to the zone. This would modify the discussed NSEC record, stating there are no other domains between `alfa.org` and `bravo.org`. Also a new NSEC-type record will be introduced, telling no domains between `bravo.org` and `charlie.org` exist. Of course, both NSEC-type records will be signed.

An attacker is now still able to replay the earlier NSEC-type resource records responses, by using any of the discussed active attacks. For example by correctly guessing the identifier, and sending a flood of DNS packets. A client will accept these responses, as long as corresponding authoritative records are not in the cache, the validity period still holds, and no new ZSKs are in use. Making the new `bravo.org` domain unavailable for clients that are target of such attack. Nevertheless, when packet injection failed, i.e. the query id or source port were incorrectly guessed, an attacker has to wait until the NSEC record is removed from the cache – which usually happens when the record's TTL expires.

Remark that this attack is not limited to modifications that result in mutation of NSEC-type records, but to any modified record. So if a data manager changes some A-type record in a zone, the old record accompanied by the old record's signature can be used in a replay attack by an attacker. Making the modified data unavailable to a client that is subject to this attack.

All these problems and flaws have to do with the offline signing process that DNSSEC has chosen. In this way it is not possible to actively interact with a client to come to a (cryptographic) commitment. Because of this choice, freshness – for example by using random nonces – of responses cannot be guaranteed. Resulting in the fact that all signatures are static according to the data they are signing.

3.5.11 NSEC Related Attacks

During the discussion of the NSEC-type resource records, it was already mentioned this record type leaks data that is supposed to be private. DNS data used to be assumed to be in principle publicly available, which was one of the reasons why confidentiality was never an objective of the DNSSEC. It is indeed the case that DNS data is publicly available, however the retrieval of data and specific sub domains are not to be publicly known. The DNSSEC protocol designers however stated that the so called NSEC-walking "is not an attack on the DNS itself" [12].

Nevertheless an extension to the DNSSEC protocol was released. RFC 5155 presents the new NSEC3-type resource record, because the NSEC-type "introduced undesired policy issues" [42]. Instead of directly exposing zone information, NSEC3 hides the zone information in cryptographic hashes. Although this seems a good solution, [6] states this makes zone enumeration particularly harder, but less detectable.

When using NSEC records, a zone can be enumerated very quick, by using the earlier mentioned NSEC walking. Notice that zone enumeration is also possible in the regular DNS. It is harder, because all possible combinations should be checked. When an attacker has the ability to use a 4Mbit/s connection, it can make around 2^{29} guesses a day. Notice however these guesses are noticeable by the data manager because of the increased traffic.

If an attacker would query for random records to a NSEC3-type capable DNSSEC server, it can retrieve the complete list of hashes. With this list, the attacker can start hashing – by doing a defined number of SHA-1 iterations – any possible combination, and compare the resulting hash with a hash in the retrieved list. Remark this is all done in an offline fashion, making it unnoticeable for any data manager. Nowadays around 2^{34} of such offline guesses can be made by an attacker per day, using an ordinary CPU. This makes it 32 times easier for an attacker to enumerate a zone compared with the regular DNS. More computing power, will even increase this factor.

The NSEC3-type has the ability to salt the plaintext input it hashes. If this salt is changed on a regular basis, this does not prevent the previously discussed attack. Because hashes are still available offline, after the global fetch iterations. The only thing that might happen is that records are removed in the meantime of the brute-force attack. Nevertheless, many of the records will still be available in most cases.

A solution to this problem is simple, however not feasible to implement in the current DNSSEC environment. It would not be a problem if denial of existence records are being signed in realtime. Responses will then just state the queried name does not exist.

3.5.12 Summary

In this subsection all attacks that have been discussed will be listed. Table 3.3 illustrates this summary. All shown attacks are listed, together with what component they attack (i.e. **c**onfidentiality, **i**ntegrity, **a**vailability, or a combination thereof). Next, it is also shown whether the attack is still possible and an explanation why. The new attacks that only apply to DNSSEC are also included.

CIA	Name	Possible?	Why?
C	Passive Attacks	✔	Still possible, because everything is sent in plaintext.
CIA	Active Attacks	✘	Not possible anymore, since DNSSEC clients will validate the signatures supplied in a response. Active messing with the protocol will therefore be detectable. Note however that availability is still of concern: the attacker can block queries and/or responses.
I	Passive Cache Poisoning	✘	Was already not possible anymore in the DNS due to in-bailiwick checking.
I	Identifier Guessing and Query Prediction	✘	Not possible anymore because of the included signatures that can be validated by any client.
I	Active Cache Poisoning	✘	See above.
A	Amplification Attack	✔	Possible and heavily effective because of the introduced resource record types that are significantly larger. Factors of 120 have been reported.
I	Hierarchical Trust	✔	Not really an attack, but a reminder that zones higher in the hierarchy have more responsibility when it comes to trust. In DNSSEC this reminder even became stronger, due to the introduction of the DS-type record, that entangles trust towards a key in lower level zones.
CIA	Other Attacks	✔	Attacks that are out of the scope of the DNSSEC are still possible, plus they can attack all facets. Remark that key material is added by DNSSEC, these should be handled (and secured) with care.
I	Timing Attacks	✔	Possible since DNSSEC needs absolute timing for its signature validation period.
I	Replay Attacks	✔	Possible because of absolute timing and the fact that DNSSEC is an offline protocol, such that a validity period is supplied with a signature.
C	NSEC Related Attacks	✔	When the NSEC-type denial of existence record is used zone enumeration is very easy, leading to confidentiality problems. Even the introduced NSEC3 record is not safe for zone enumeration.

Table 3.3: *Summary of all attacks on DNSSEC including their effectiveness*

Chapter 4

DNSCurve

Both the regular DNS and its security extension (DNSSEC) have been introduced. However, it appears not all the security vulnerabilities the regular DNS suffers are actually solved by DNSSEC. Even some new attack vectors have been introduced by DNSSEC. Therefore another extension that helps in securing the Internet's naming system will be introduced in this chapter: DNSCurve.

This chapter starts again with a general introduction, followed by the (short) history of DNSCurve. Next the objectives will be discussed, and afterwards the protocol will be specified in detail. The chapter continues with listing all earlier discussed attacks, although now alignment with DNSCurve. Finally a comparison will be made between DNSSEC and DNSCurve.

4.1 Introduction

The entire design process of DNSSEC took (considering the NSEC3 RFC being the last one) over 15 years. And in the end, it appears that not all known attacks specific for the DNS can be prevented, detected, or stopped by DNSSEC. This lead to the introduction of DNSCurve, which tries to eliminate all vulnerabilities that are known for the standard DNS.

4.2 History

Not much is known of DNSCurve's history besides that it is designed by Daniel Bernstein. Next to his diverse scientific work as a mathematician and cryptologist, Bernstein is known for his popular open source software, such as djbdns, qmail, and ucspi-tcp. These were written respectively to replace at that time popular naming software (BIND), email software (sendmail), and Internet daemon (inetd). All of these have suffered substantial security holes every now and then.

Bernstein's software is always written with security in mind. For his qmail project, he rewarded $500 to the first person who could find "a verifiable security hole in the latest version of qmail" [18]. The latest version (1.03) was released mid 1998. The offer however still stands.

His DNS software suite djbdns [17] – consisting of several DNS related programs, such as a separate authoritative name server called tinydns, and a caching name server called dnscache – got the same guarantee as qmail, only with an offer of $1,000. The latest version (1.05) of djbdns was released at the start of 2001. This offer was rewarded to Matthew Dempsky at the beginning of 2009. He managed to find a security hole in the way the djbdns package handled outgoing DNS responses over TCP [24].

4.3 Objectives

This section will explain the objectives of DNSCurve, and also – just as important – the non-objectives. Like most of this chapter, this text is based on the DNSCurve's official website [6].

As mentioned in this chapter's introduction, DNSCurve tries to mitigate all known vulnerabilities that are specific to the DNS. DNSCurve tries to contribute to every part of the 'CIA'-principle, which is discussed in the upcoming subsections.

4.3.1 Confidentiality

DNSCurve will protect confidentiality by encrypting all DNS queries, and responses. This means no eavesdropper (or: sniffer) will be able to deduct information in the communication between a DNS client and a DNS server. It should however be noticed this guarantee can only be given on data in the DNS part of a packet. I.e. all surrounding headers, like the IP and UDP or TCP ones, will still be in plaintext. Meaning that an attacker is still able to deduct some information, like when a query is sent, how many queries are sent, and how long a query is. Nevertheless, more detailed information is kept hidden from the attacker, such as the query name.

4.3.2 Integrity

In the regular DNS nothing is done to protect data from being altered by an adversary somewhere in the middle of transport. DNSSEC's main objective is to protect the DNS' integrity. DNSCurve protects the integrity of packets by cryptographically authenticating all the queries and responses. Besides authenticity of packets, DNSCurve also provides a special form of non-repudiation. This means that a client or server cannot deny to each other, they did sent a query or response in combination with their public key. More on this later.

Not truly integrity related, but DNSCurve should be backwards compatible with the regular DNS. Meaning that if a host is not DNSCurve capable, it should still be able to use DNSCurve enabled hosts.

4.3.3 Availability

The regular DNS has the ability to increase availability of server side, by using more authoritative name servers that are serving the same zone data. If one of these name servers is unavailable, a client will try another one that is an authority for the zone. Because both DNSSEC and DNSCurve are extensions to the standard DNS, both also still support this feature.

Availability of the regular DNS's client side is harder to achieve. Because integrity concerned attacks can influence availability behavior. If an attacker is able to change a zone's referral, this could make this zone unavailable. DNSCurve will not accept rogue DNS data, because of the integrity (or better: authenticity) guarantee from the previous subsection.

4.3.4 Non-Objectives

The DNSCurve website [6] does not particularly specify what DNSCurve is not able to achieve. It is however worth mentioning some – maybe even implicit – things DNSCurve will not solve. In the previous chapters it is explained what vulnerabilities the regular DNS suffers. The DNSSEC chapter discussed what vulnerabilities were solved, but also what remained open, i.e. could not be solved by DNSSEC.

DNS information is seen by its protocol designers as public information. Indeed, everyone is able to query anything to a particular authoritative name server. If records exist for this query name, it will respond. This does however not mean that a data manager would like clients to know the contents of an entire zone. This information is usually revealed to a client, by zone enumeration. It should be noticed this will always be possible, also when DNSCurve is used. Because a server has to respond to a query, either positively or negatively. If it does not answer to negative query results (tinydns does this for example), this will still give a client the idea something's wrong. For instance because a previous query did get a positive response, meaning a timeout implicitly would indicate a negative response.

In the DNS chapter's security section availability was discussed from several points of view. One of which was the case that the authoritative name servers were unavailable due to an availability attack, for example caused by a DoS attack. DNSCurve will not protect against any of such attacks, other than the standard DNS already does. (For instance by being able to specify more than one authoritative name server for a zone.)

The previously discussed traffic amplification attack remains possible when DNSCurve is used. However, DNSCurve is not burdened by the large amplification factors that some of DNSSEC's resource records produce.

4.4 Specification

Now that the objectives of DNSCurve are clear, it is time to look at its specification. That is done in this section, by first explaining the protocol itself. Next, a roundup of all used cryptographic primitives is given, followed by the way DNSCurve handles key material. Furthermore, a DNSCurve traversal will be shown, together with a description of regular DNSCurve topologies. Finally a subsection is dedicated to what will change for data managers, when using DNSCurve. Most of this text is taken from [6] and [30].

4.4.1 Protocol

The DNSCurve protocol is – just like DNSSEC – an extension to the standard DNS protocol. This means it adds functionality to the DNS protocol, without interfering with the protocol's early workings. In this subsection the protocol itself will be explained. This is done by first introducing the two protocol formats DNSCurve distinguishes. Furthermore the realtime nature of the protocol will be discussed. Finally the term 'link level security' will be introduced and explained.

Protocol Formats

To be as efficient with bandwidth as possible, DNSCurve introduced its own protocol format, that can house regular DNS packets. However, to be backwards compatible a standard DNS protocol format had to be introduced too. In this way, DNSCurve packets would be able to be transfered through already existing firewall configurations, that for example check packets that pass on UDP port 53. Downside of this approach is that a little more information is revealed to a possible attacker.

The format that implements the bandwidth efficient solution is – appropriately – called the 'streamlined format'. The backwards compatible version is named the 'TXT format'.

Streamlined Format
The streamlined format is an efficient protocol format, that houses all the

information that is needed for a DNSCurve conversation. It should however be noticed there is a difference between a streamlined query, and a streamlined response.

Let's first focus on a streamlined query. Figure 4.1 illustrates what a streamlined query looks like. Remark that the figure is 16-byte horizontally aligned, one box in the protocol format is one byte.

```
                                          1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| Q| 6| f| n| v| W| j| 8|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|               CLIENT PUBLIC KEY               |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|           CLIENT NONCE          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                                               /
/              CRYPTOGRAPHIC BOX                /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 4.1: *DNSCurve streamlined query format*

A query starts by an 8-byte fixed string (i.e. `Q6fnvWj8`) that will always prefix a streamlined DNSCurve query packet. What follows is the client's public key, that is 32-bytes long. How and what this client public key exactly is, will be discussed in the next subsection, that focuses on the used cryptographic primitives.

Next, a 12-byte client selected nonce can be distinguished. This nonce should be different for each and every packet. It is used to guarantee freshness and helps by preventing replay attacks. The specification [6] suggests to use either a counter or time related data for the first 64-bits of the nonce. The last 32-bits can be cryptographically random generated. This would make collisions improbable.

Finally an arbitrary length cryptographic box is found. This cryptographic box will contain the entire original standard DNS query packet, only now encrypted, and authenticated. What this cryptobox exactly is, will be discussed in a later subsection, that focuses on DNSCurve's cryptography.

It might seem awkward there is no length field, specifying the size of the cryptographic box. This is done on purpose, because operating systems will deliver the entire streamlined packet together with a total number of bytes, to a program. Making it easy to calculate the only variable length in the packet.

Notice that all bytes discussed here, are truly bytes. In the sense that there is no restriction on (alpha-)numeric character use.

Now focus on a streamlined response. The format of a streamlined response packet is portrayed in Figure 4.2.

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| R| 6| f| n| v| W| J| 8|
+--+--+--+--+--+--+--+--+--+--+--+--+
|           CLIENT NONCE            |
+--+--+--+--+--+--+--+--+--+--+--+--+
|           SERVER NONCE            |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                                               /
/               CRYPTOGRAPHIC BOX               /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 4.2: *DNSCurve streamlined response format*

Just like the query format, this packet is prefixed with an 8-byte fixed string (that is equal to the one from the query, only the first byte is different, indicating it is a response: `R6fnvWj8`). What follows is a copy of the client's 12-byte nonce, that was sent to the server in the query packet. Next, a server selected 12-byte nonce is included. This nonce, together with the client's one, will guarantee total freshness, and should be created in the same matter as the client nonce was. Finally, the cryptographic box containing the entire regular DNS response can be found. The length of the box is again arbitrary.

It is obvious that the client's public key is not needed anymore in a response, since this is already known by the client. It is therefore intentionally left out.

TXT-format

It is clear that a firewall that actively checks any packet that is passing on UDP port 53 will notice a streamlined query is not a true DNS query. Therefore it will reject the packet from passing, making DNSCurve unavailable to clients behind this firewall. That is why the TXT-format was introduced. As the name suggests, DNSCurve uses the TXT-type resource records to 'hide' its details in. Just like with the streamlined format, a differentiation has been made between a query and response format.

There is not much space in a regular DNS query to hide any information in. Most firewalls allow only one query in the query section, and zero records in the answer-, authority-, and additional section. This means that all information that a client has to send to a server, should be put inside a part that has an arbitrary length. It is obvious this can only happen inside the query section's query name. Figure 4.3 illustrates how a TXT-format query looks like. This figure is again 16-byte aligned. Notice that every byte in the following figures that consists of two characters should be seen as a hexadecimal value.

```
                                   1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  ID |00|00|00 01|00 00|00 00|00 00|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                                               /
/               DNSCURVE QUERY NAME             /
/                                   +--+--+--+--+
/                                   |01 00|00 01|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 4.3: *DNSCurve TXT-format for queries*

Notice the first 12 bytes are the DNS header, that is known from section 2.3.5. The first two bytes are the query's identifier, that is selected by the client in a cryptographically random fashion. What follows are two bytes representing the bit fields, that respectively indicate this packet is a standard query, not an authoritative answer, not truncated, no recursion is desired (and available), and nothing is wrong with the response. The next four 2-byte numbers indicate how many queries, regular-, authoritative-, and additional answers are included in this query respectively. As it is a query, only one question is included. The query name that follows has (just like a query name in a regular DNS query) an arbitrary length and a special format, more on that in the next paragraph. Finally, two 16-bits values can be found. The first byte indicates this is a TXT-query, while the second byte states it has to look in the `IN` class.

Now focus on the query name. From the regular DNS chapter it is known a query name (and response names in a resource record) consist of labels, that all have a maximum length of 63 bytes. All of these labels are preceded by a respective size byte, and the entire query name is terminated with the null-label.

To be also able to send all the information that is carried with a streamlined query packet in a TXT-format query packet, some special encoding is needed. Figure 4.4 illustrates how a query name looks like, in a TXT-format query (and response) packet.

The first block contains an encoded version of both the client selected nonce, as well as the cryptobox. The encoding is needed, because the query name can – according to RFC 1035 [45] – only consist of letters, digits, and hyphens. So if the entire space that a byte offers was used (i.e. using binary),

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                                               /
/    ENCODED CLIENT NONCE + ENCODED CRYPTOBOX   /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|36| x| 1| a|                                   /
+--+--+--+--+                                   /
/            ENCODED CLIENT PUBLIC KEY          /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                                               /
/                   NAME OF ZONE             +--+
/                                            |00|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 4.4: *DNSCurve query name for TXT-format queries and responses*

unknown characters would be included, violating the RFC. Therefore all binary data, like the nonce, the cryptobox, and also the client's public key have to be encoded. DNSCurve has chosen a base-32 encoding. This means that binary data will only be represented by using 22 letters and digits. The key usage subsection will introduce base-32 in more detail.

Remark that the base-32 block of both the client nonce and the cryptobox should consist of labels that have a length of maximally 50 bytes. Also notice that every of these labels should be preceded by a length byte.

What follows after the nonce and cryptobox is the client's public key that is again base-32 encoded. An encoded public key is 51 bytes long, and prefixed with the 'magic string' `x1a`, resulting in a 54 byte (i.e. `0x36`) label.

Finally the name of the zone that is queried is attached. Notice this would probably correspond to the query name inside the cryptobox' DNS packet. The entire query name is suffixed with the null-label. The zone name is included to be seen more like a true DNS conversation, and therefore increasing the chance of passing through a firewall.

It is obvious that this will occasionally violate RFC 1035's limit on query names, that was set on 255 bytes. This will happen when the original DNS packet's query name contains over 100 characters. The 512-byte limit on DNS packets that RFC 1035 specified is also likely to be exceeded sometimes. DNSCurve aware hosts should however be able to handle such packets and query names.

A TXT-format response is a reply to the query packet that is portrayed in Figure 4.3. How the response looks like, is illustrated in Figure 4.5.

Notice that the first part is almost a one-on-one copy of the query packet. Only two things have changed: the bit field (i.e. `0x84`) states this packet is

```
                              1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  ID |84|00|00 01|00 01|00 00|00 00|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                                               /  }
/              DNSCURVE QUERY NAME  +--+--+--+--+  } QUERY
/                                   |01 00|00 01|  }
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|c0 0c|01 00|00 01|00 00 00 00|RDLEN|           /  }
+--+--+--+--+--+--+--+--+--+--+--+--+           /  }
/                                               /  } ANSWER TXT RR
/        RDATA (SERVER NONCE + CRYPTOBOX)       /  }
/                                               /  }
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 4.5: *DNSCurve TXT-format for responses*

a response, and also an authoritative answer. Secondly it tells this packet contains one resource record in the answer section, besides one query. The entire query section is a direct copy of the query that initiated this response, nothing is changed there. So the content of this section is exactly like the one depicted in Figure 4.4.

What follows is a resource record (formatted as discussed in section 2.3.5). The first two bytes indicate the query's name. This might look awkward, because no characters can be seen. The first byte (i.e. `0xc0`) however, indicates DNS compression is used. DNS compression has not been introduced, but in short this means a DNS capable host should look for the query name after 12 (i.e. `0x0c`) bytes beyond the packet's start. This is directly after the header, and it can be seen this is the query name, in the query section. Meaning the response name of this resource record is the query name.

What follows is the type (TXT) and class (`IN`) of the resource record. Next, four bytes indicate the TTL of this record is zero, suggesting a caching name server is not allowed to cache this record. The two bytes after that (`RDLEN`) indicate what the size of the `RDATA` section is. The `RDATA` section itself consists of the server's nonce and the cryptobox, that contains the standard DNS response packet. Notice this is binary data, so no encoding is needed, nevertheless – just like query names – TXT records should also be split up in labels, having a maximum size of 255 bytes each. Remark that firewalls will accept binary data in the `RDATA` section.

Realtime

One of the items that are quickly noticed, are the included nonces. As mentioned, nonces guarantee freshness of a packet, and in particular for the cryptobox. It might seem strange that nonces are included in the plain, but that is their function. Because of the so called 'avalanche effect', adding arbitrary data inside the plaintext, will result in a totally different ciphertext. Nevertheless, to check authenticity of the nonces (and implicitly of the entire packet), they are to be included both outside the ciphertext (in plaintext), and inside.

Remark that because of the usage of nonces, this directly implies that DNSCurve uses realtime encryption and authenticity mechanisms. And not – like DNSSEC does – offline signing of data. Because of this choice, lots of the problems that DNSSEC suffers are directly solved. Such as replay attacks, complex denial of existence records, usability when it comes to resigning of data, and there is no validity period in which authenticity can be guaranteed.

The biggest objection against realtime encryption in DNSSEC was the concern about speed and implicitly stability. DNSCurve tries to solve this by using new high speed cryptography primitives. This, together with the previously discussed protocol and a new, easier to deploy, architecture.

Link Level Security

In the security section of this chapter, a more detailed comparison between DNSCurve and DNSSEC will be given. It is however good to already see one of the major differences between both extensions.

First some terminology is needed. By looking back at a regular DNS topology, every host through which a DNS conversation takes place (i.e. a stub resolver, caching name server, and an authoritative name server), is able to alter a DNS packet. This is one of the main causes that initiated the DNSSEC project, because integrity of a DNS conversation could not be guaranteed. Every host in the middle can alter DNS, such host does not even have to be concerned with DNS. In a way, a caching name server does already influence a DNS conversation. It could respond a cached RR, while the authoritative name server has a new, updated version of the RR available. A client can never determine whether the response it received originated from the caching name server's cache, or not.

DNSSEC added a mechanism to enable a client to indicate what kind of answers it would like to receive. If it for example would like to verify signatures on its own, it can set the `CD`-bit in the query. In this way all signatures, keys, and such will be send along. This means that in DNSSEC end-to-end security can be established, because every type of host in a DNS topology can verify signatures on its own, if it wants to. In short, DNSSEC does really protect the DNS data's integrity.

When focusing on DNSCurve, some other observations can be made. DNSCurve does not introduce any new resource record types, indicating it does not focus on the DNS data itself. Instead, it first concentrates on transport of the DNS data, and second implicitly on the data. This means DNSCurve only guarantees hop-by-hop security. Meaning that a stub resolver can use DNSCurve to securely retrieve a response from a caching name server, although this caching name server will return a different result, than the responsible DNSCurve capable authoritative name server would give. Verification of data retrieved by an initial DNSCurve response is therefore not possible by end nodes, if intermediate steps were done.

This phenomenon is sometimes also referred to as link level security. Meaning that only the communication link itself is secured, independent of all semantic data that is transferred over it. Another indication of this behavior is the fact that public keys in DNSCurve are not specific to zones (like in DNSSEC), but to name servers. During the DNSCurve key usage description and a traversal, all this will become more clear.

4.4.2 Cryptographic Primitives

In the DNSSEC chapter also a subsection was dedicated to introduce the used cryptographic primitives. As earlier mentioned, DNSCurve uses a new high speed cryptography suite to be able to deliver realtime cryptographic responses to queries. This subsection will introduce all the individual parts of this suite.

When looking back at the objectives, the cryptographic suite should fulfill two general requirements. It should supply confidentiality and integrity (both authenticity, and a special form of non-repudiation). Because each of these requirements are solved by different cryptographic primitives, DNSCurve uses a combination of primitives. In the paragraphs below, each of these individual primitives will be discussed. First the entire project in which the new high speed primitives have been developed is introduced.

NaCl and CACE

To accommodate the ease of implementation and use of new primitives, the NaCl project was initiated [7]. The goal of the NaCl (pronounced ‘salt’, and an abbreviation of Networking and Cryptographic library) project is to provide “all of the core operations needed to build higher-level cryptographic tools”. Besides specification and implementation, the NaCl project also validates and verifies the primitives it includes.

The project – part of a bigger European Commission funded project CACE (Computer Aided Cryptography Engineering) – provides a high-level interface for all cryptographic primitives it supports. The new primitives that DNSCurve uses, and that are implemented by NaCl are the elliptic curve Diffie-Hellman CURVE25519 primitive, the stream cipher SALSA20, and the message authentication code POLY1305.

Since there are no other libraries that support the newly defined primitives used by DNSCurve, it is straightforward to see why NaCl will be used. By using NaCl, also other features it brings can be leveraged. One of the benefits is the automated selection of the fastest primitive for a specific platform, because NaCl has the ability to support several implementations of the same cryptographic primitive. One implementation will for example be faster on an x86 than on a SPARC CPU architecture. Different compilers and compiler options may also play a part – in addition to hardware differences – in the speed of a cryptographic primitive. Therefore during the installation of NaCl several different implementations, compilers, and compiler options will be analyzed performance-wise. In the end the fastest combination of implementation, compiler, and compiler options will be used.

Other advantages of NaCl are the low memory footprint of the library itself, and the fact that there are implementations that resist side channel attacks. Additionally, due to NaCl's modular setup it can also support already established cryptographic primitives, like the symmetric ciphers DES, AES, and asymmetric ciphers such as RSA, Diffie-Hellman, ElGamal. The cryptographic hash functions MD5 and SHA-256 are implemented as well[1]. Although none of these primitives are used by DNSCurve, it is good to know it will be relatively easy to implement a new cipher suite for DNSCurve.

Each of the primitives have their own function, the up following paragraphs will discuss the primitives that DNSCurve will use. Most of this text is based on both [6] and [23].

Diffie-Hellman

The Diffie-Hellman key exchange is widely known nowadays. It was developed by Whitfield Diffie and Martin Hellman in 1976 and can be considered one of the first public key cryptography systems. The name 'key exchange' indicates the protocol is used to establish a 'shared secret' between two parties. This shared secret will be used as a key for faster secret key cryptography later.

DNSCurve makes use of a new Diffie-Hellman variant, that uses elliptic curve cryptography. CURVE25519 was designed by Daniel Bernstein at the beginning of 2006 [20]. The name is derived from the prime $2^{255} - 19$ that is the base for the elliptic curve's finite field. The CURVE25519 primitive implements a Diffie-Hellman key exchange, only much with much stronger security guarantees than the older discrete logarithm problem. Besides upgraded security, it is also faster than current key exchange protocols. In comparison with RSA, it has also the advantage of much smaller key lengths. All CURVE25519 keys (both private and public) have a length of 255 bits,

[1]Nevertheless, at the moment only the mentioned new cryptographic primitives, the secret key authentication HMAC primitives, and the SHA-256 and SHA-512 hash functions have been implemented.

meaning they can fit inside 32 bytes. Elliptic curve keys of this bit length supply a security that is comparable to that of 3,000-bit RSA keys [6].

ECDH (Elliptic Curve Diffie-Hellman) generally works as follows. Sender Alice chooses a cryptographically random secret key a of 32 bytes. She uses the CURVE25519 base primitive to generate a 32-byte public key A from this secret a. Receiver Bob has already done the same, meaning he has a random secret key b, and a corresponding public key B. Their shared secret is generated by doing a scalar multiplication with the private key, and the other person's public key. So the 32-byte shared secret that Alice uses is aB, the one of Bob – that is equal to Alice's – is bA.

The final result of the Diffie-Hellman key exchange will later be used as a symmetric input key for both the stream cipher and message authentication code.

Stream Cipher

Generally, symmetric key systems are a lot faster than asymmetric key systems. This is why people switch back to symmetric cryptography after a key exchange. DNSCurve does the same, by using the SALSA20 stream cipher. This cipher brings confidentiality for DNSCurve.

The SALSA20 stream cipher was designed by Bernstein at the end of 2007 [21]. It is a stream cipher that needs a 256-bits key, and a 64-bits nonce as input, together with a data position of 64-bits. The number 20 points at the number of rounds the stream cipher uses. Although there are variants that use less rounds, such as SALSA20/8 and SALSA20/12, that use 8 and 12 rounds respectively. Using less rounds degrades security, but improves speed.

DNSCurve uses a slightly different variant of SALSA20: XSALSA20. The only difference between both systems is that XSALSA20 uses a 192-bits nonce. The implication of this choice can already be seen in DNSCurve's protocol format: the client and server nonce are together 24 bytes long.

Message Authentication Code

A key exchange and encryption are one thing, an attacker could however flip one bit of the cipher, resulting in a totally different plaintext, without the receiver of the message knowing the message was altered. Indeed, one of the design requirements for DNSCurve is integrity, something that a message authentication code (MAC) can fulfill.

DNSCurve uses a variant of the POLY1305 MAC. This MAC is designed by Bernstein at the beginning of 2005 [19]. The name is derived from the prime $2^{130} - 5$ that is used as a polynomial evaluation modulo factor. The standard version of POLY1305 makes use of the symmetric block cipher AES.

POLY1305 computes a 16-byte authenticator from an arbitrary length message, a nonce and a (shared) secret key. The nonce and authenticator can be included (even in plaintext) with the message. The recipient of the message can now verify the message by generating an authenticator on its own. If the generated authenticator and the authenticator are equal, the message can be considered authentic.

When looking at DNSCurve's implementation of POLY1305, it is known the XSALSA20 stream cipher is used. The POLY1305 primitive therefore uses the XSALSA20 cipher instead of AES. This saves some intermediate calculations, such as generating separate AES keys, and doing AES encryptions.

Cryptobox

All the individual primitives have been briefly introduced by now. The NaCl project does however give the ability to use all the primitives in one big operation. The resulting cipher of this operation is called a 'cryptobox'. It is designed "to meet the standard notions of privacy and third-party unforgeability for a public-key authenticated-encryption scheme using nonces" [7].

The cryptobox is a combination of the CURVE25519 elliptic curve Diffie-Hellman key exchange, the XSALSA20 stream cipher, and the POLY1305 message authentication code and is designed by Bernstein. Its precise description can be found in [23].

It is clear by now that confidentiality is met by using the XSALSA20 cipher, in combination with the key exchange. Integrity, and especially authenticity, is guaranteed by the POLY1305 primitive. However, in the introduction of this subsection it was mentioned that 'a special form of non-repudiation' is needed too. This 'special form' points at the fact that authenticators are made explicitly for a recipient. Meaning that a receiver will only be able to verify the message, using the shared secret it has with the sender. Notice this is a different function than public key signatures (that DNSSEC uses) fulfill. A public key signature can be verified by anyone that can obtain the corresponding public key.

Bernstein speaks of "public key authenticators" [7] indicating that only the sender and receiver can verify the authenticators. So no other party than the two participants can authenticate a cryptobox packet. When an adversary would alter the cryptobox, only the participants will find out. Remark this also implies that a participant cannot show any proof to a third party that it received a certain packet from the other participant. Since it could have generated the authenticator by itself. This will be called 'limited non-repudiation', because only the two participants can guarantee non-repudiation of each other's packets.

The input of the cryptobox operation is the arbitrary length message, a 24-byte nonce, and the 32-byte shared secret. The output cipher will be as long as the input message's length, only with a 16-byte authenticator added to it.

4.4.3 Key Usage

In the DNSSEC chapter it was obvious how the public keys are retrieved, due to the introduction of the DNSKEY-type resource record. As mentioned, DNSCurve does not introduce such new resource record, although it was unclear how a DNSCurve client could obtain a public key. That is what is discussed in this subsection.

During design it was chosen not to introduce a new resource record type, mainly because "new record types are a continuing source of interoperability problems (...) and would require extensive software upgrades" [6]. Therefore DNSCurve came up with a smart trick, to be able to include public keys without changing the standard DNS protocol's behavior. From the link security explanation, it is clear that public keys are specific to name servers, instead of being specific to zone data. This makes it possible to 'hide' the public key of a name server in the name server name of a NS-type record.

Take the `example.org` domain, that has the name server `ns1.example.org` in the regular DNS environment. A DNSCurve capable variant of this name server would be:
`uz5dkhm9g380kyx9slmktyvmb1h0ck7whwzc5uqvl8f1cwfp8zl3ub.ns1.example.org`
This is the base-32 encoded variant of the public key of the `ns1.example.org` name server. The encoding is prefixed with the magic DNSCurve string `uz5`, the 51 suffixing bytes represent the 32-byte public key. Remark that this magic string differs with the one from the client public key (`x1a`). This magic string therefore defines the public key of a server public key.

According to RFC 1035 [45] the full range of a byte can be used to represent a domain name. However, this would again lead to interoperability, therefore it was chosen to base-32 encode all public keys that house in legacy fields of the DNS protocol. A domain name for a name server being one of them.

During a traversal, whenever a client sees a label of a name server name that is exactly 54 bytes long and starts with the magic string `uz5`, it will threat this name as a DNSCurve public key. So when it contacts this name server, it will use the public key that was in the name server's domain name. During an actual traversal that is done in the next subsection, this will become more clear.

Base-32 Encoding

Until now, the base-32 encoding has been mentioned quite some times, it was however nowhere officially introduced.

The base-32 encoding (and corresponding encodings such as base-16- and base-64 encoding) have been introduced to represent binary data in alphanumeric characters. It works relatively simple, by grouping binary data in groups of 5 bits – in the base-32 case. These 5-bits represent one character from a 32 character alphanumeric alphabet. This alphabet includes all digits and letters, except `a`, `e`, `i`, and `o`.

For example the hexadecimal number `0x89ab`, looks like this in binary: `1000 1001 1100 1101`. Splitting it in groups of 5 bits: `01101 01110 00010 00001` and aligning each of these bits to a character in the alphabet, we get the base-32 encoding: `df21`.

Remark that the base-32 encoding that is used in DNSCurve is not compatible with the encodings described in RFC 4648. The alphabet of DNSCurve differs from that in the RFC.

4.4.4 Traversal

All preliminary knowledge to understand DNSCurve is introduced by now, therefore it is time to do a DNSCurve traversal. Just like the ones in the DNS and DNSSEC chapter.

Likewise, also here some assumptions are made. It is assumed the root name servers are *not* DNSCurve capable. This means that trusted keys for particular name servers are needed, to initiate trust at some point. Such point is called a 'trust anchor'. It should be remarked this traversal is purely educational, therefore some details have been simplified.

Just like in the regular DNS and DNSSEC traversal examples, a client wants to resolve `foo.example.org`. Only this time it has to use DNSCurve when possible. Figure 4.6 illustrates the entire DNSCurve traversal, notice the yellow circles that point out the order in which a query or response is made. In the list below these circles will be discussed respectively.

0. Just like in the DNSSEC traversal, also a DNSCurve capable client should have a trusted starting point. Because DNSCurve was not designed by any committee or team of the IETF, there is a really small chance the root servers will ever implement DNSCurve[2]. This means a trusted anchor should start the authenticated chain (to use terminology from the DNSSEC world) somewhere under the root. Remark that by storing such anchor locally, trust is indirectly encapsulated by this decision. In this example the `.org` name servers are DNSCurve capable. To make this known to a client, only the NS-type resource records are needed.

   In the figure two different DNSCurve capable name servers (notice the magic `uz5` string) for the `.org` domain can be distinguished. Also, the client has a keypair. The upper base-32 encoded string is the client's

[2]Nevertheless, Matthew Dempsky did write a draft RFC describing DNSCurve in detail [30]. It was published at the beginning of 2010. The draft expires August 2010.

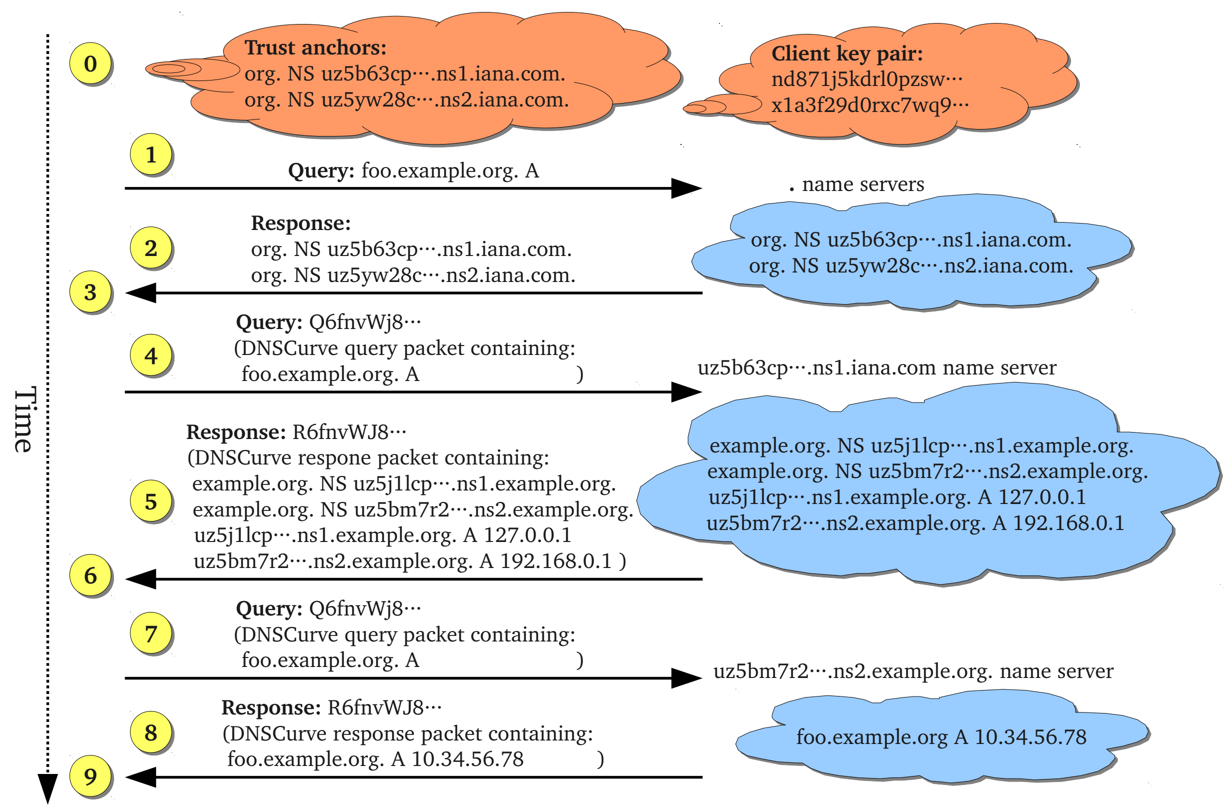
0
Trust anchors:
org. NS uz5b63cp···.ns1.iana.com.
org. NS uz5yw28c···.ns2.iana.com.
Client key pair:
nd871j5kdrl0pzsw···
x1a3f29d0rxc7wq9···
1
Query: foo.example.org. A
. name servers
2
Response:
org. NS uz5b63cp···.ns1.iana.com.
org. NS uz5yw28c···.ns2.iana.com.
3
org. NS uz5b63cp···.ns1.iana.com.
org. NS uz5yw28c···.ns2.iana.com.
Query: Q6fnvWj8···
4
(DNSCurve query packet containing:
foo.example.org. A)
uz5b63cp···.ns1.iana.com name server
Time
Response: R6fnvWJ8···
(DNSCurve respone packet containing:
example.org. NS uz5j1lcp···.ns1.example.org.
5
example.org. NS uz5bm7r2···.ns2.example.org.
uz5j1lcp···.ns1.example.org. A 127.0.0.1
6
uz5bm7r2···.ns2.example.org. A 192.168.0.1)
example.org. NS uz5j1lcp···.ns1.example.org.
example.org. NS uz5bm7r2···.ns2.example.org.
uz5j1lcp···.ns1.example.org. A 127.0.0.1
uz5bm7r2···.ns2.example.org. A 192.168.0.1
Query: Q6fnvWj8···
7
(DNSCurve query packet containing:
foo.example.org. A)
uz5bm7r2···.ns2.example.org. name server
Response: R6fnvWJ8···
8
(DNSCurve response packet containing:
foo.example.org. A 10.34.56.78)
9
foo.example.org A 10.34.56.78

Figure 4.6: *Graphical representation of a DNSCurve traversal*

private key. The lower base-32 encoded string is the client's public key, that is preceded by the magic (client public key) string: `x1a`.

1. Because no trusted keys are found for the root (`.`), a regular DNS A-type query for `foo.example.org` is send to one of the randomly picked root servers.

2. The root servers do not know anything about `foo.example.org`, but it does know something about the `.org` zone. A regular DNS response is received, with this information included.

   Note this response can be subject to active cache poisoning attacks. However, because a trusted key is used for the `.org` zone, a rogue response will be noticed in the next step.

3. The first 'verification' step can now start. Since no DNSCurve response is received, nothing has to be done with the received packet. However, referrals have been received that refer to the same zone that is in the trusted keys of circle 0. To initiate trust in these keys, the received ones and the already known ones are compared. If they equalize, the

trusted chain is started here.

Because two referrals were received, one is randomly selected. The name server `uz5b63cp···.ns1.iana.com` is chosen and apparently it contains a DNSCurve public key. The key will be extracted and a shared secret will be generated, using the client's private key and the just retrieved public key. This shared secret can be cached by a client, to be used again when approaching this same name server.

4. A DNSCurve query has to be sent to the `uz5b63cp···.ns1.iana.com` name server. A streamlined DNSCurve query packet will be made ready. Besides the static prefix string, the client's public key will be inserted, together with a client selected 12-byte nonce. The cryptobox contains a ciphered version of the same query as the one sent in circle 1.

5. The DNSCurve capable name server received a streamlined DNSCurve query packet. In this packet, the client's public key is included, as well as the client nonce. With the client's public key, and its own private key, the server can generate the same shared secret that the client generated before. Using the nonce and the shared secret, the server is able to unpack the DNSCurve query, and read the boxed regular DNS query. With the normal DNS query packet, it is business as usual.

   `foo.example.org` is not found at the name server, it does however know where to find the `example.org` zone. It has two name servers for `example.org`, and because both are in-bailiwick with the domain itself, the authoritative name server will also include glue records. These glue records indicate at what IP addresses the two name servers can be found. Notice that this is regular DNS behavior, DNSCurve did not influence this.

   Now that there is an answer, the server can start a response. Since the server received a DNSCurve query, it will also respond using DNSCurve. The streamlined response packet will contain the repeated client nonce, a 12 byte selected server nonce, and the cryptobox that contains the entire response packet.

6. The client sent out a DNSCurve query, so it should also receive one. Therefore all non-DNSCurve responses for this query, will not be accepted. Remark this can only be found out, if the cryptobox is opened.

   The response was a DNSCurve response, so it will decrypt[3] the response using the shared secret, and the client- plus server nonce. If

[3]Using the terms 'encrypting' and 'decrypting' is not entirely right. The cryptobox does not only provide confidentiality, also authenticity and limited non-repudiation. Nevertheless, for convenience both terms will still be used together with the synonyms 'boxing' and 'unboxing'.

this unboxing goes well, it will threat the unboxed response like a regular DNS response.

Two NS-type records were returned, together with their corresponding glued addresses. One of the two is randomly picked: `uz5bm7r2···.ns2.example.org`. This name server domain name again includes a DNSCurve public key. The key will be extracted, and again a shared secret will be generated, using the client's private key and the server's public key.

7. The same query as before will be send to the selected name server. Although now boxed using the shared secret, and a fresh client nonce. The query that is boxed, is again the same one as sent in item 1.

8. The authoritative name server of the `example.org` zone receives a DNSCurve query. It will generate the shared secret, and afterwards try to open the cryptobox. If this unboxing succeeds, it can read the regular DNS query and the name server will act upon it.

   The name server has a corresponding result for the query, and will respond with a streamlined DNSCurve response packet. The cryptobox is generated by using the response itself, the shared secret, the received client nonce, and the created server nonce. In this packet, the A-type resource record response can be found.

9. The client sent out a DNSCurve query, so it expects to receive a DNSCurve response. That is the case, so the cryptobox is unboxed using the shared secret, and the included server nonce – the received client nonce can be compared with the one sent earlier. If decryption goes okay, a regular DNS response appears, that includes the right response. Stating that `foo.example.org` is housed at `10.34.56.78`.

Notice that during this traversal everything went okay, i.e. all name servers responded, every query was positively answered, all DNSCurve queries got a DNSCurve response, and all cryptoboxes decrypted fine. However, for each problem that might occur scenarios have been specified in the regular DNS how to deal with this. If a DNSCurve query is not answered with a DNSCurve response, the response should be dropped. The same should be done when the cryptobox could not be opened. For the simplicity of this text however, all these details were intentionally left out.

Remark that for non-DNSCurve hosts, the traversal is still possible. These hosts will see the DNSCurve name server names just as normal (but lengthy) name server domain names.

4.4.5 Topology

Previous subsections gave a good introduction to DNSCurve and its capabilities. This subsection will discuss the DNSCurve topology by aligning it with the topology that is known from the DNS and DNSSEC chapters. Nevertheless, DNSCurve distinguishes one new entity: the forwarding name server. This name server can be compared with a proxy server, known from the HTTP world. It simply forwards an incoming DNS query towards another DNS capable server, the response of this server is than fed back to the actual client. This behavior is interesting to deploy DNSCurve in a coordinated way.

Until now, only the topology of Figure 2.4 has been discussed. DNSSEC's topology did not change something to it, the main question there was where to verify the signatures. DNSCurve does alter the topology a bit by introducing the forwarding name server. Still the question is open where to encrypt and decrypt messages.

From the previous chapters, it is known a caching name server has two sides: the name server and the resolver. DNSCurve distinguishes four different caching name server roles: a full DNSCurve capable resolver, an outgoing DNSCurve resolver, an incoming DNSCurve resolver, and a non-DNSCurve resolver.

A full DNSCurve capable resolver can use DNSCurve on both its name server side as well as on its resolver side. An outgoing DNSCurve resolver only uses DNSCurve on its resolver side, so it is able to talk DNSCurve with authoritative name servers. An incoming DNSCurve resolver can speak DNSCurve only on its name server side, so to actual clients or stub resolvers. A non-DNSCurve resolver is not able to understand any DNSCurve conversation, on either side.

In the DNSCurve topology, stub resolvers and actual clients are grouped together and illustrated as clients. These clients can be DNSCurve, and non-DNSCurve capable. Furthermore the distinction is made between non-DNSCurve authoritative servers, and DNSCurve capable ones. With these nine different entities, a good overview of a DNSCurve topology can be given.

Figure 4.7 illustrates the DNSCurve topology. The discussed entities can be seen as boxes. The lines that connect each of the boxes indicate what entities communicate with each other. The color and shape of the lines show whether this connection is secure (green and solid) or not (red and dashed).

Clients can ask queries to each of the entities. It is obvious non-DNSCurve clients cannot have any secure communication with an entity. Curve enabled clients on the other hand, can securely contact all DNSCurve capable entities (although not with the outgoing resolver).

A caching resolver usually contacts authoritative name servers, it can however also contact the forwarding name server. Remark that a non-DNSCurve forwarder has been left out, since it would have the same connections as a non-DNSCurve authoritative name server.

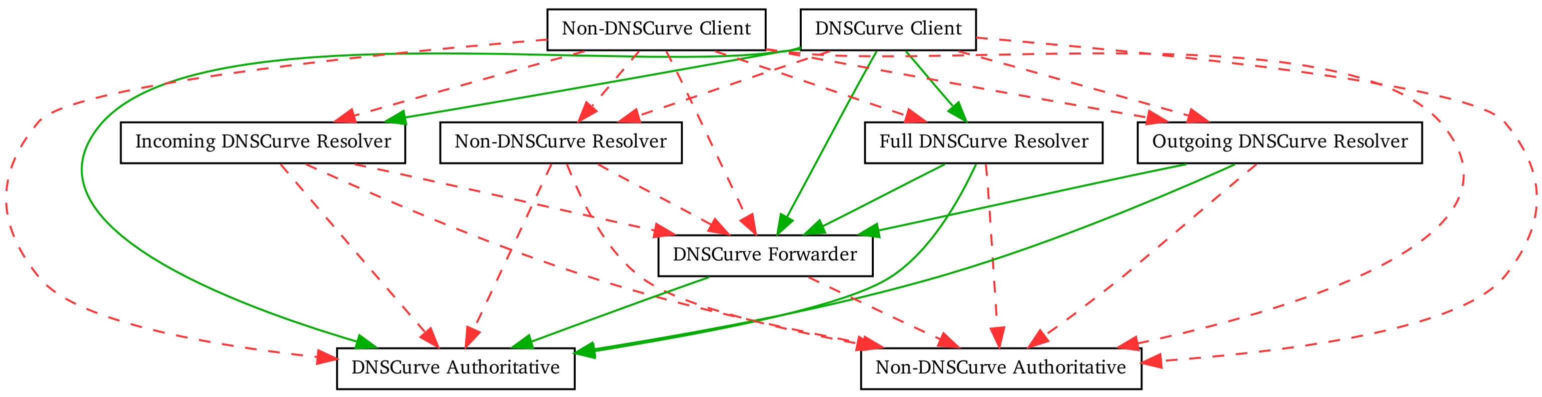

Figure 4.7: *Graphical representation of the DNSCurve topology*

The forwarding name server holds contact with authoritative name servers. If the forwarder is able to speak DNSCurve on its incoming side (i.e. the side that has contact with the resolvers), it can forward regular queries to authoritative name servers that are not DNSCurve capable. In this way deployment can be drastically made easier, because no current authoritative configurations (e.g. server, management, and labor wise) have to be adopted. Just putting a forwarder for each name server would do the job. Implications for data managers will be discussed in the data manager section. Notice that using a DNSCurve enabled forwarder in front of a DNSCurve capable authoritative name server is actually superfluous, nevertheless it shows the hop-by-hop security an entire DNSCurve topology would give.

When looking purely locational, the top six entities usually reside on an organization's LAN. Or, when considering private setups, the client is at a person's home while the resolver is at the ISP. The bottom three are regularly also closely located to each other. For both situations this is done to decrease latency, and increase stability due to possible dropped UDP packets.

The figure also portrays when a DNSCurve packet will be boxed and unboxed. Namely, at every entity. Take the longest entire secure path, so a path in which every line – from client to authoritative name server – is green. Both entities in the middle (i.e. the full resolver and the forwarder) will two times encrypt, and two times decrypt a DNSCurve packet. The endpoints – the DNSCurve client and the authoritative server – will both one time encrypt, and decrypt the packet. This is indeed an overhead, but the new high-speed cryptographic primitives should be able to handle this work, even on regular machines and CPUs.

In an ideal situation all Internet clients should have an entire secured path – the whole path from client to authoritative name server should be solid and green. This is an unrealistic goal, in the writer's opinion not even DNSSEC will reach this stage in the coming 10 years. Nevertheless, coming close to an ideal situation is possible. The most important link to protect, is the one between the resolver and any authoritative source that resides on the WAN. Firstly, since most clients do not directly contact authoritative servers, instead they use a local caching name server. Secondly, clients – especially stub resolvers – exist in so many flavors, developing a DNSCurve variant for all of them is undoable work without community support. Thirdly, because the link between the client and the resolver is in many cases relatively secure, since it resides on the LAN. And finally, since this link usually crosses many routing devices, it is more likely to be intercepted by an adversary.

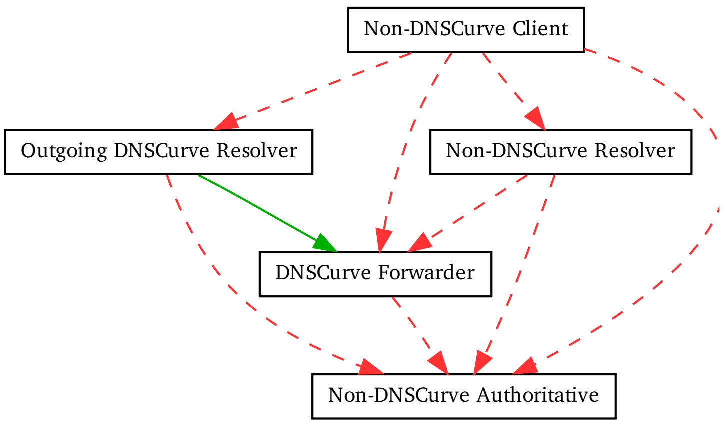

Figure 4.8: *Graphical representation of a desired DNSCurve topology*

So to start deployment of DNSCurve, it is desired to focus on implementing a DNSCurve capable resolving server (either full, or at least outgoing) and a forwarding name server. At the start of this master project, no implementation existed that could do any of the DNSCurve enabled jobs. Although along time, more and more code came available. The desired situation of the topology after this master's project is sketched in Figure 4.8. This would mean the most important part of a DNS conversation can be protected, i.e. the link between the resolver and the authoritative source.

4.4.6 Data Managers

The DNSSEC chapter dedicated a special subsection to what is about to change for DNS data managers, when DNSSEC would be deployed. This same question is answered by this subsection, only this time in relation with DNSCurve.

In order to be called DNSCurve capable on the authoritative side, the server has to be able to accept both streamlined and TXT-format queries. To let a client know what the public key of the server is, a data manager should have generated a keypair for this server. From the key usage subsection it is known the public key can be obtained by a client using the NS-type referral. Remark that the approach of key management is drastically different, in comparison with DNSSEC. DNSSEC uses one (although usually more) key per zone, while DNSCurve uses one key per name server.

When the data manager generated a keypair, the generator program returns the base-32 encoded public key (prefixed with the magic `uz5` string), and the private key. Note that for the name server name containing the public key also an address (i.e. a A- or AAAA-type) record should be made. Also notice this process should be repeated for every name server of the zone.

The public key has to be made public by submitting it to the zone one level up in the hierarchy. This zone is usually maintained by a so called RRR (registry, registrar, or registrant). Notice the DNSCurve specification [6] does not mention how the public key should be transfered to such registry, however it is clear that integrity should be contained. Most of these automated systems already have the ability to do this in a cryptographically secure fashion (by using the EPP protocol).

Now that the referring name server is able to send the right referral containing the public key, it is time to focus on the authoritative name server itself. The easiest solution for a data manager is to implement a forwarding name server, 'in front' of the current authoritative name server. Notice in this way almost no changes have to be made on the administrative backoffice system of the authoritative servers. The forwarder could even be implemented at the same physical machine that hosts the authoritative name server, however using a separate machine for this purpose is recommended, because the most CPU intensive tasks will be done by the forwarder (i.e. encrypting and decrypting packets).

Another option would be to implement DNSCurve by using a new DNSCurve capable authoritative name server. Although none exist at the moment, this would also imply the administrative backoffice of an organization would need a change. Such system took years of becoming what it is for an organization nowadays, meaning that data managers think twice before making this transition. It would however help if an upgrade of the current used authoritative name server software would make it DNSCurve capable.

Nevertheless, in all cases a data manager can gently move to a full capable DNSCurve authoritative environment. Because of the availability feature of the DNS, most zones have their zone data served by more than one name server. Since DNSCurve is name server based – instead of zone based – a transition to DNSCurve can be done stepwise. For example, first only a zone's third name server will be DNSCurve capable. This would mean only 33.33% of all queries could be DNSCurve capable. If this works out well, more name servers of the zone can make the change. Notice this is not a feature of DNSCurve itself, this does however not make is less usable for this situation.

In contrast with DNSSEC, afterwards deployment of DNSCurve, not much has to been done by a data manager. DNSCurve does not need periodical resignaturing, and relatively quick key rollovers. This does not mean DNSCurve does not need key rollovers at all. None of the DNSCurve specifications mention what an average key rollover length would be. However [6] states that "an attacker who spends a billion dollars on special-purpose chips to attack CURVE25519, using the best attacks available today, has about 1 chance in 1,000,000,000,000,000,000,000,000,000 of breaking CURVE25519 after a year of computation". General rule of thumb therefore is to roll over a name server's key once every year.

Key rollovers are relatively easy to deploy. Generate a new keypair, notify the up level's data manager, and inform the authoritative name server or the forwarder about the new keypair. This same procedure can even be used for an emergency key rollover, for example because the private key is exposed.

This subsection mainly focused on the authoritative side. Still, the recursive side is just as important. Although a transition in this area is – just like in the DNSSEC world – relatively easy, by just replacing a caching name server with a DNSCurve capable one. This is possible because normally there is not a backend in use that controls the caching name server. There is no need for key management or any other administrative task for such caching name server. Because it is able to generate a random keypair when the server is started. What however could be needed is a trust anchor.

DNSSEC suffered quite some increase in memory, bandwidth, and CPU usage. Memory usage will only grow constantly when using DNSCurve. However, the more memory available the better. Because the biggest performance boost for a DNSCurve capable host can be achieved by caching shared secrets. Since this is the most intensive operation of a DNSCurve conversation. Notice that caching name servers will consume a bit more (regular DNS) cache space, due to increased length of NS-type records.

Bandwidth usage will also increase, although again only constantly. The output of the cryptobox primitive has the same size as the input, only the 16-byte authenticator is added. The streamlined format will also add a constant size (with a maximum of 52 bytes). Remark that when the TXT-format is used, more bytes – this is however still constant – are needed.

The only thing that will increase on a non-constant base is CPU usage. Since every packet should be encrypted and decrypted. The next chapter will give more detail in resource usage by DNSCurve.

4.5 Security

The DNS chapter introduced all known attacks to the DNS that are known until this date. Both DNSSEC and DNSCurve are developed to minimize the number of vulnerabilities of the DNS protocol. The DNSSEC chapter discussed all known DNS attacks, although now in the DNSSEC world. Eventually it seemed DNSSEC introduced some new attack vectors itself, that were also explained. The same will be done in this section. It will also discuss whether the specific DNSSEC attacks apply to DNSCurve. Furthermore, any vulnerability that DNSCurve might bring will be explained.

4.5.1 Passive Attacks

Line sniffing by an attacker is still possible when DNSCurve is used. Although now, the attacker will not be able to deduct any information regarding the DNS packet itself. This means the entire DNS packet will be hidden from the attacker, by using a confidentially secure cryptographic primitive (i.e. the XSALSA20 cipher).

Keep in mind however that an attacker will still be able to see the IP and UDP or TCP header that prefixes the DNS packet. This means an attacker can still determine the packet is a DNS packet (due to the source- or destination port), what host the packet is originating from and destined for (due to the source and destination IP), and how long a packet is (and therefore implicitly able to deduct the length of a query, when the packet was a query). If a counter was used as the first 64 bits of a nonce any sniffer can also see the traffic rate. Besides, an adversary can conduct behavioral analysis, i.e. when clients are active, how active they are, etcetera. Remark that analysis of a TXT-format DNSCurve packet is a bit easier for an attacker, in comparison with the streamlined format. The TXT-format also gives away more information, since the name of the zone that is queries is attached to the query name – that contains all DNSCurve information.

Just like in the DNSSEC chapter, here also the remark that whenever a regular DNS conversation takes place, none of the security guarantees explained here can be achieved.

4.5.2 Active Attacks

Active attacks can actively influence a communication line. A DNS conversation that makes use of DNSCurve is not vulnerable to active attacks that focus on confidentiality or integrity.

Because an active attack is equal to a passive attack, only with more capabilities. All that is said on passive attacks also applies to active attacks. Meaning that confidentiality of the query name is protected by DNSCurve (this can however only be fully guaranteed when the streamlined format is used).

When looking at integrity a DNSCurve conversation is entirely protected. If an attacker would flip some bits of a DNSCurve packet, this will be noticed by any of the participating parties. Because the 16-byte authenticator will not validate, and therefore the query or response will be discarded. Notice this only applies to the nonces, public keys, and the cryptobox. If other parts of a streamline or TXT-format packet are altered, behavior is unknown. Although in most cases the packet will be discarded, leading to availability issues.

If an attacker would insert a regular DNS response after a DNSCurve query, a participant in the DNSCurve protocol will immediately drop this false packet. It will keep waiting for the right response, until a certain time-out is reached.

Nevertheless, just like DNSSEC, DNSCurve cannot guarantee anything on availability. If an attacker would block a DNSCurve query or response. A client will still think the server is unavailable.

Again, if no DNSCurve conversation is used, nothing can be guaranteed regarding security. Things are as worse as the DNS chapter described in the security section.

Due to the hierarchy that heavily influences DNSCurve's behavior, this last observation has implications for the authenticated chain that DNSCurve can offer. The hierarchical trust subsection of this section will go into more detail regarding this 'attack'.

4.5.3 Passive Cache Poisoning

Passive cache poisoning was already not possible anymore in the regular DNS, due to in-bailiwick checking. Let's focus on the case where DNSCurve is used and in-bailiwick checking would not exist.

These kind of attacks were actually initiated by an authoritative source. Looking back at the example, the (rogue) data manager of `example.org` stated that `www.google.com` lived at a certain address. Because DNSCurve only protects the communication channel, it will not be able to detect such attacks. Anything that is put in an cryptobox will receive integrity protection, even if this would be false information. Semantically DNSCurve puts no restriction on what it will encrypt or decrypt. This is a good observation of the channel security that DNSCurve guarantees.

DNSSEC on the other hand did also not protect against these attacks, since no signatures would be supplied for non-authoritative data. This convention of not signing non-authoritative data is a good example of a semantic rule, implying the object security guarantee of DNSSEC. Due to this rule, all non-authoritative data can still be altered by an attacker when using DNSSEC, this would however still need an active attack.

Therefore the in-bailiwick check is still needed in the DNSCurve environment.

4.5.4 Identifier Guessing and Query Prediction

The first attack that actively interfered with the DNS protocol workings, was at some point still possible when using DNSSEC. DNSCurve does protect against this kind of attacks.

When the streamlined format is used, an attacker will not be able to deduct any information from the DNS packet itself. As this will all be hidden inside the cryptobox. So it cannot predict the packet's identifier. Nevertheless, it can still guess the query name, type, and class. Because a client can still be directed towards an attacker owned website, that shows (non)-existing images at the target domain.

However, to be able to build a correct rogue response packet, the attacker should get hold of the server's private key, or the conversation's shared secret. Due to the security guarantee of the CURVE25519 primitive this is infeasible. So any rogue response will be directly discarded by a client, if it is unable to open the cryptobox for whatever reason. The replay attack subsection will show that even replaying valid responses, will not work.

When the TXT-format is used, the attacker is able to see the identifier of the outer packet. Notice this identifier should be different from the identifier of the packet that is boxed inside the cryptobox. If an attacker would be able to inject a TXT-format response and the outer identifier would match the query one's, the packet will still be discarded. Since the attacker is unable to generate a valid cryptobox without the server's private key or knowing the used shared secret.

Something that should be observed is the security involved with entities that unbox a query or response, and sent along the corresponding regular DNS packet. A DNSCurve forwarder is such entity, but also recursive name servers do this. See Figure 4.8 and notice the insecure connections between a forwarding name server and a non-DNSCurve authoritative name server, and also between the client and the resolvers.

A DNSCurve forwarding name server receives a DNSCurve query, containing a DNS packet with a certain identifier. If it would unbox the DNSCurve packet and directly pass the regular DNS packet along to its authoritative name server, the forwarding side of the forwarding name server is subject to an identifier guessing and/or query prediction attack. Since an attacker can inject rogue responses originating from the authoritative name server. Therefore any entity that acts in the middle of a DNSCurve conversation should at least replace the identifier with a cryptographically random one, before it passes the packet through. This should be done especially when non-DNSCurve packet are passed along.

Not using DNSCurve for a DNS packet of course implies that these kind of attacks are still possible.

4.5.5 Active Cache Poisoning

Active cache poisoning uses a lot of ideas from the identifier guessing and query prediction attack. Therefore all text that discussed DNSCurve in relation with query injection attacks still applies.

Meaning that an attacker is unable to insert rogue responses when DNSCurve is used. Because a DNSCurve client will not accept DNSCurve responses where the cryptobox cannot be opened.

When a participant in a DNSCurve conversation is unable to open a cryptobox, this can have several reasons. Firstly, the used shared secret (and thus implicitly the participant's own private key or the other party's public key) appears to be wrong by whatever reason. Secondly, the used nonce to encrypt or decrypt the box appears to be rogue, for example due to unreliable transport. Thirdly, the authenticator of the packet appears to be different, this can happen when bits were accidentally- or deliberately flipped by purpose, during transport. So any attacker tempering with DNSCurve packets will never succeed.

The same statement still applies also with active cache poisoning attacks: not using DNSCurve still gives an attacker the opportunity to carry out these kind of attacks.

4.5.6 Amplification Attack

The amplification attack will be discussed for the third time now. Both DNS and DNSSEC are 'vulnerable' to this type of attack. DNSSEC provides however resource record types that are much more likely to be used by attackers. In the security section of the DNSSEC chapter it was shown that enormous amplification factors could be reached when DNSSEC was used.

During discussion of bandwidth usage of DNSCurve, an answer has already been given on how DNSCurve would influence this type of attacks. Since DNSCurve does not introduce any new resource record types, an attacker should therefore use the regular DNS protocol in order to abuse DNSCurve's bandwidth usage. Although, it has been shown already in the data manager's subsection that DNSCurve does only add a constant number of bytes to a DNSCurve conversation. While DNSSEC for example could add a linear factor, due to the output of a cryptographic signature operation. DNSCurve only added small constant portions: the 16-byte authenticator, the public key in a query, the public key hidden in a NS-type referral, a 24-byte nonce, and some protocol string overhead.

In general DNSCurve is therefore as effective as the regular DNS is for amplification attacks. This means DNSCurve does not open new attack vectors, at least not with an eye on the factors that already existed in the standard DNS.

4.5.7 Hierarchical Trust

Hierarchical trust was, and is an important notion in both DNS and DNSSEC. It is also of significance in DNSCurve.

Just like in DNSSEC, the 'authenticated chain of trust' should be rooted somewhere in order to put trust in a DNSCurve answer. It has already been mentioned that chances are very little that root servers, and even cc- or gTLD name servers will ever going to support DNSCurve. This has severe implications for trustworthiness of a DNSCurve conversation. To be able to fully trust a DNSCurve answer, all resolve iterations should be using DNSCurve. This can never happen, when the root servers and TLD name servers do not support DNSCurve.

For example, if an attacker is able to forge a response from the `.org` name servers, stating that `uz5fr3p···.ns.attacker.example.com` is the name server for `example.org`, instead of the real DNSCurve enabled name server `uz510vb···.ns1.example.org`, nothing can be guaranteed by a DNSCurve client. I.e. it will never know that the `attacker.example.com` is not the real name server of `example.org`. Notice the forged name server does not even need to be DNSCurve capable.

A solution for this problem is to enforce trust by supplying trusted keys for DNSCurve enabled zones. It is however undoable to manually add trust for each and every zone that is using DNSCurve. Currently no system has been developed to solve this problem. Also, the DNSCurve specifications do not indicate how to enroll a trusted key for a certain zone. In the traversal example of this chapter an own format was used, i.e. the NS-type record that should have been present at the zone's parent.

However, due to DNSCurve's design it is not the case that DNSCurve is worthless at a zone when a parent zone does not support DNSCurve. Since a name server's public key is hidden in the name server name, the parent zone will implicitly support DNSCurve by serving the domain name containing the public key. This differs from DNSSEC. In fact, DNSCurve can be deployed bottom-up instead of top-down, like DNSSEC would. This can greatly improve the adoption of DNSCurve. Since data managers can secure their own zones, without waiting for parental data managers to be DNSCurve capable.

Nevertheless, it should be clear that hierarchical trust is still of great concern in the DNSCurve world.

4.5.8 Other Attacks

Other attacks were attacks outside the scope of the DNS itself. It is clear DNSCurve does not mitigate any of these attacks, they are therefore still possible.

There are however some cases that should get some more attention, when using DNSCurve. Just like DNSSEC, also DNSCurve capable hosts use key material for their operations. DNSSEC hosts could use HSMs to securely store their private keys. However, because HSMs do not support the cryptographic primitives that DNSCurve uses, special care should be taken with this key material.

If an attacker has possession over a participant's private DNSCurve key, it is able to entirely control this participant's behavior. If the participant is a server, it can inject anything in a response. To be prepared for such attacks, good (emergency) key rollover schemes should be in place.

4.5.9 Timing Attacks

Until now, only attacks have been discussed that the regular DNS suffers. The following subsections are attacks that were introduced by DNSSEC. Although this time they are aligned to DNSCurve, to see whether DNSCurve also suffers them or not. Starting with timing attacks.

Because DNSSEC introduced absolute time, timing information is of great value in a DNSSEC environment. Biased time greatly influences the behavior of DNSSEC capable hosts. It can for example make signatures invalid, while they are not. Leading to the so called DNSSEC suicide.

DNSCurve on the other hand does not rely on time at all. Hosts using DNSCurve do not need to have a reliable time source. Just like the regular DNS did not need one. Because all times are – just as in the regular DNS – relative again. Notice that an unreliable timing source could however influence cache effectiveness, because TTLs would be reached earlier or later.

4.5.10 Replay Attacks

In the DNSSEC world replay attacks have a great impact. When an attacker would save lots of different responses from a DNSSEC capable host, it can replay these messages to clients as long as the signature's expiration is not reached. Such response can for example include a denial of existence record, stating there are no records between `alfa.org` and `charlie.org`, while in the meantime `bravo.org` was created. Making the `bravo.org` domain unavailable for clients subject to this attack.

As mentioned, the nonces used in DNSCurve guarantee freshness of a packet, and in particular for the cryptobox. It might seem strange that nonces are included in the plain of DNSCurve packet, but that is their function. Because of the so called 'avalanche effect', adding arbitrary data inside the plaintext, will result in a totally different cipher. And since both parties (i.e. the client as well as the server) add their own nonce, total freshness can be guaranteed, even if one of both parties turns out to be dishonest.

If at least one party uses a distinguishable way of generating nonces, this means a DNSCurve capable host is immune to replay attacks. The DNSCurve specification [6] recommends to use the first 64 bits of a one-side nonce to be either a counter, or a derivate of the current time. While the remaining 32 bits can be filled with cryptographically random data.

The total 96 bits nonce will therefore guarantee this nonce will not be used again. Remark however this guarantee should only hold in a conversation with the same host. The nonce 1 can for example be used for different hosts, however it is not allowed to be used again in a conversation with same host.

Notice that time skews can influence the behavior of a DNSCurve capable host. Therefore the specification [6] states this clock is not allowed to jump back, or at least it should not be noticeable inside the nonce. Such that freshness can still be guaranteed.

Due to the freshness guarantee, even queries to the same host with the same query name will always give a different cryptobox if a different client nonce is used. If the client would however use the same nonce, the query packet cryptobox will be the same. While the response packet on the other hand will be different no matter what, because of the included by the server picked nonce.

Therefore, replaying of old response packets will not succeed, or the client should have used the same nonce, towards the same host, with the same query name, packet identifier, type, and class. The chance this will occur is negligible.

4.5.11 NSEC Related Attacks

Because DNSCurve uses realtime encryption and decryption, there is no need for a denial of existence record. This means that DNSCurve is as vulnerable to zone enumeration attacks as the regular DNS is.

The DNSSEC subsection on this attack, already mentioned that [6] tells an attacker with a 4Mbit/s connection was able to do around 2^{29} guesses on a regular DNS name server per day. Remark however, this are noisy guesses, i.e. they are detectable by a data manager that closely monitors traffic.

Also mind that an attacker better uses the regular DNS protocol when doing these guesses to a DNSCurve capable host, as it will cost more CPU resources to constantly decrypt the received responses. (Encryption is not needed, as the client can send the exact same packet to the name server, with the same client nonce, public key and the cryptobox. Only the response will vary, due to the chosen server nonce.)

In short, DNSCurve is just as good or as bad as the regular DNS is, regarding zone enumeration.

4.5.12 CPU Exhaustion Attacks

Finally a subsection that neither the DNS chapter, nor the DNSSEC chapter includes. This subsection focuses on CPU exhaustion attacks. Attacks that try to exhaust a DNSCurve capable host's CPU. In most cases this host will be the name server. In fact this type of attacks can be seen as an availability attack, since exhausting the CPU would either slow down a host or maybe even make it unreachable.

The fact this is the first time this attack appears has to do with the real-time fashion of DNSCurve. DNSCurve uses relatively CPU intensive (cryptographic) operations, operations that both the DNS and DNSSEC do not need. Due to these operations, an attacker could send floods of valid DNSCurve queries. A DNSCurve capable name server will need CPU power to unbox the query, and return a response after boxing it again. Notice however unboxing is the hardest operation, since this is the first time the shared secret will be computed (by using the CURVE25519 primitive). When the response is sent, the shared secret will be used again implying this time only a symmetric cryptographic operation is needed (the XSALSA20 cipher plus the POLY1305 primitive to compute the MAC) to make the cryptobox.

So to exhaust a CPU, the attacker should use a different client keypair for each query. In this way the server will constantly have to compute new shared secrets. Remark that an attacker can pre-compute all these queries offline before sending them, giving it a head start. And if the source IP address is spoofed in a random matter, it is almost impossible for the target name server's data manager to stop this attack.

The next chapter will focus on shared secret caching algorithms. Nevertheless, a cache is finite so it will also exhaust at some point, leaving a good attack vector for adversaries.

DNSCurve is one of the first algorithms that uses public key cryptography via UDP. Therefore not much corresponding information regarding such attacks is available. The widely used HTTPS protocol does not suffer the exact same attack, mainly because it uses TCP making IP spoofing impossible. The next chapter will discuss this in more detail, together with benchmarks.

4.5.13 Summary

In this subsection all attacks that have been discussed will be listed. Table 4.1 illustrates this summary. All shown attacks are listed, together with what component they attack (i.e. **c**onfidentiality, **i**ntegrity, **a**vailability, or a combination thereof). Next, it is also shown whether the attack is still possible and an explanation why. The attacks that apply to DNSSEC are also included, together with the new attack that solely applies to DNSCurve.

CIA	**Name**	**Possible?**	**Why?**
C	Passive Attacks	✘	Not possible, at least not for DNS specific information due to secrecy primitive.
CIA	Active Attacks	✘	Not possible, DNSCurve clients will detect when rogue responses are no responses are included. Note however that availability is still of concern: the attacker can block queries and/or responses.
I	Passive Cache Poisoning	✘	Was already not possible anymore in the DNS due to in-bailiwick checking.
I	Identifier Guessing and Query Prediction	✘	Not possible anymore because of the integrity, authenticity, and special non-repudiation guarantee of the used cryptobox primitive.
I	Active Cache Poisoning	✘	See above.
A	Amplification Attack	✔ / ✘	Possible, although factors are comparable with the regular DNS, which is much less than known factors when DNSSEC is used.
I	Hierarchical Trust	✔	Not really an attack, but a reminder that zones higher in the hierarchy have more responsibility when it comes to trust. For DNSCurve this still applies, because implicitly a higher zone delivers the public key of a name server lower in the hierarchy.
CIA	Other Attacks	✔	Attacks that are out of the scope of the DNSCurve are still possible, plus they can attack all facets. Remark that key material used by DNSCurve should be handled with care.
I	Timing Attacks	✘	Not possible, DNSCurve does not need absolute time.
I	Replay Attacks	✘	Not possible, DNSCurve guarantees total freshness by using two-side nonces and acting realtime upon a query. Meaning that every DNSCurve packet inside a DNSCurve conversation will be different.
C	NSEC Related Attacks	✘	Just as good or as bad as zone enumeration was in DNS. DNSCurve does not introduce any denial of existence records, it will deny existence in realtime.
A	CPU Exhaustion Attacks	✔	Possible, although the next chapter will give some more detail on this attack and its effectiveness.

Table 4.1: *Summary of all attacks on DNSCurve including their effectiveness*

4.6 Comparison with DNSSEC

So far lots of differences between DNSSEC and DNSCurve have been discussed, mainly by aligning the way both protocols mitigate security related attacks. This section will summarize many of the differences that have already been discussed, but it will also explain some not earlier covered dissimilarities.

What to Protect

The protocol subsection in the specification section introduced the biggest difference between the two protocols. DNSSEC focuses on protecting the 'object', while DNSCurve protects the communication 'channel'. This is shortened to guaranteeing 'object security' and 'channel security'. In general it means that DNSSEC is more involved with the semantics of the DNS protocol than DNSCurve is. It prescribes rules what to sign, and what not. While DNSCurve will protect anything that is send over the line. This also implies DNSSEC can guarantee end-to-end security, since the protected objects will be passed along by intermediate DNSSEC capable hosts. Due to the signatures that accompany the object, a recipient will always be able to detect if something was accidentally- or deliberately modified during transport. DNSCurve on the other hand will only guarantee hop-by-hop security, meaning that every intermediate DNSCurve capable host is able to alter the object without the recipient knowing. Because all the hosts in the middle will encrypt and/or decrypt the cryptobox.

Something that illustrates this difference well, is the way keys are used inside DNSSEC and DNSCurve. Every zone (i.e. object) in DNSSEC has its own keypair, while DNSCurve only needs one keypair per DNSCurve host.

Objective Differences

In the objectives section of this chapter more significant differences between DNSSEC and DNSCurve came to light. It was explained that DNSCurve does guarantee some confidentiality, at least on a DNS packet itself. DNSSEC does not offer any confidentiality, i.e. every packet is still send in plaintext.

Both DNSSEC and DNSCurve provide integrity and in particular authenticity. Because DNSSEC uses public key signatures, any party can verify the accompanied signature DNSSEC delivers. DNSCurve on the other hand uses public key authenticators. This implies that no third party can verify the authenticators included in a DNSCurve packet, since the authenticator is based on the shared secret between both participants. Even the participants cannot proof to a third party it received a particular packet from the other participant, because it could have generated the authenticator by itself.

When it comes to availability, DNSCurve gives some added protection because it can immediately detect forged responses (i.e. when the crypto-box fails to open) leaving no opportunity to attack availability by supplying false referrals for example. Since DNSSEC does not sign non-authoritative data, attacks focusing on these non-authoritative resource records are still possible. Remark that both protocols are still vulnerable to denial of service attacks, such as UDP floods. Besides, DNSSEC also gives new opportunity to the already known amplification attacks. Because DNSSEC introduced new resource record types that are considerably larger than already existing types a DNSSEC capable authoritative name server is a good source to initiate such attack. DNSCurve does not add significant length to its packets, making it as good or as bad as the regular DNS in regard to these type of attacks.

Realtime and Offline

DNSSEC is, in comparison with DNSCurve, an offline protocol. In the sense that all cryptographic operations are done in an offline matter. This has as an advantage that serving of zone data is just as intensive as in the regular DNS – although more bandwidth will be used. It has however also a big disadvantage, since it creates the opportunity for an attacker to conduct replay attacks. Due to the offline nature every signature generated by DNSSEC is accompanied with a validity period. Within this period, the signature appears to be valid.

DNSCurve on the other hand is a realtime protocol. It performs all cryptographic operations directly when it receives a DNSCurve packet or response. Disadvantage of this approach is that an attacker can make a DNSCurve host able to degrade its service, by exhausting its CPU. Advantage on the other hand is that replay attacks are not possible, due to the freshness guaranteed by two-side nonces. Meaning that even when one party is dishonest, freshness will still be ensured.

Resource Usage

When DNSCurve is used, storage needs of a DNSCurve capable host will not change drastically. Only a small number of number of bytes are added because of the public key that is hidden inside NS-type records. In contrary, DNSSEC does increase storage needs linearly. Factors between 4 and 12 have been reported [5]. This is caused by denial of existence records, and the fact that every RRset (including these denial of existence records) needs a cryptographic signature. The length of these signatures is specified by the used cryptographic suite. DNSSEC is designed in such a way that every cipher suite can be used, although at this point mainly the RSA/SHA-1 suite is in use [51]. DNSCurve can also support different cryptographic suites, it currently however only supports the CURVE25519, XSALSA20, and POLY1305 combination. This combination is referred to as the 'cryptobox'. Benefits of the elliptic curve used by DNSCurve is its speed, plus smaller key- and cipher lengths. This is the reason why the public key of a DNSCurve host can fit inside a single domain name label.

The strength of the 255-bit public key used in DNSCurve can be compared with a 3,000-bit RSA key. However, 3,000-bit RSA would not be fast enough to be used inside a realtime cryptographic DNS protocol [6].

Amount of Work

When looking at the amount of work for a data manager, most of the work has to be done to get DNSSEC deployed and maintained. The software should be DNSSEC compatible, key- and zone signing keys should be generated and administered, and all zone data has to be modified to include signatures, denial of existence records, and key material. The key material should also be made known to a parental zone data manager. Note that these were all tasks to be performed initially, DNSSEC however also requires periodical attention. Such as resignaturing of zone data, and key rollovers. Although more and more tools [10] become available to simplify this process, it still demands good understanding from the data manager of what he is doing.

DNSCurve on the other hand is relatively simple to deploy. The used software should be made DNSCurve ready. Furthermore only one keypair per server should be generated and administered. The public key of this pair will be send towards a parental zone, although nothing in this process should be changed because of the fact the public key is 'hidden' inside the referral name. DNSCurve involves – besides key rollovers, that usually take place once a year – no periodical tasks.

The transition from being non-DNSSEC ready to fully DNSSEC capable should be done instantly. There is no intermediate step. DNSCurve on the other hand offers a gentle move to being fully DNSCurve capable. This can be done by moving all authoritative name servers of a zone step by step.

Specification and Standardization

When the specification of both protocols is compared, it appears that DNSCurve is described in only one (draft) RFC [30] while DNSSEC needed three [12] [13] [14] – leaving out the by some organizations mandatory considered NSEC3 RFC [42]. Not that quantity says a lot, the complexity of the DNSCurve protocol is in comparison with DNSSEC much more simplistic. DNSCurve does not introduce new resource record types, and does not interfere with the protocol's semantics (aside from the public key detection inside referrals). Besides, due to the complexity of the DNSSEC protocol and its additional RFCs, implementation of DNSSEC capable software will be harder and more error prone.

DNSSEC already is a true standard granted by the IETF. While DNSCurve is still in the process of becoming one. Mainly because of this standardization, DNSSEC is at the moment being deployed on a worldwide scale. Especially because since July 15, 2010 the root servers have become DNSSEC capable. Chances are very little that DNSCurve will ever be supported by the root servers. Even support by major TLDs seems unrealistic at this point. Because there is no current DNSCurve implementation, deployment of DNSCurve has not started yet. Next, DNSCurve is only supported by a select group of people, while DNSSEC is supported by a big community. Even commercial support is available.

Advantage of DNSCurve is that it can be used even when parental zones are not DNSCurve capable. In this way all zones served by a DNSCurve capable authoritative (or forwarding) name server will be protected, only of course if a DNSCurve client is used. Although it should be noted that a referral served by a non-DNSCurve capable parental zone is still vulnerable to attacks. The next section will give some more detail on this regard, by combining DNSSEC and DNSCurve in two different ways.

4.7 Combining with DNSSEC

This section tries to unite DNSSEC and DNSCurve in such a way that the best of two worlds can be combined. Theoretically seen, DNSCurve gives the most protection against all known attacks. Therefore DNSCurve will form the base of the first presented combined solution. This solution will try to use DNSSEC as less as possible, but it will nevertheless guarantee an entirely ‘secured’ traversal. The final subsection on the other hand will introduce a fully combined solution.

The term ‘secured’ needs some more explanation. In this section ‘secure’ means data integrity secure, in the sense that no cache poisoning attacks would be successful during the entire traversal. Note however this claim only holds when the client is DNSSEC *and* DNSCurve capable.

4.7.1 DNSCurve Bottom-Up – DNSSEC Top-Down Approach

The biggest disadvantage of DNSCurve at this point is the fact that it is not deployed worldwide. As mentioned, this implies the first iterations of a traversal cannot be secured using DNSCurve, because the root servers and TLD name servers do not support DNSCurve at this moment. Therefore, benefiting the security DNSCurve guarantees over an entire traversal is impossible, since attackers are still able to attack these first iterations. I.e.

referrals could be directed to attacker owned name servers, completely ignoring any security guaranteed by DNSCurve.

So to guarantee an entirely secured traversal, from top to bottom of the DNS tree, a combined approach with DNSSEC is needed since DNSSEC is already deployed and supported by the root servers and many TLD name servers nowadays. However, since DNSSEC does not sign non-authoritative data – such as referrals – DNSSEC is required up to and including the first zone that is DNSCurve capable. This subsection shows a solution in which DNSSEC usage will be limited to a minimum.

Figure 4.9 shows a traversal with a combined DNSCurve bottom-up and DNSSEC top-down approach. Note that in the figure many details have been simplified or left out. I.e. no DNSKEY-type records are requested, no glue records for the name servers are included, contents of the DS- and RRSIG-type records are not given, and DNSCurve queries and responses are still readable. Nevertheless, the up following description should give the global idea of this combined approach.

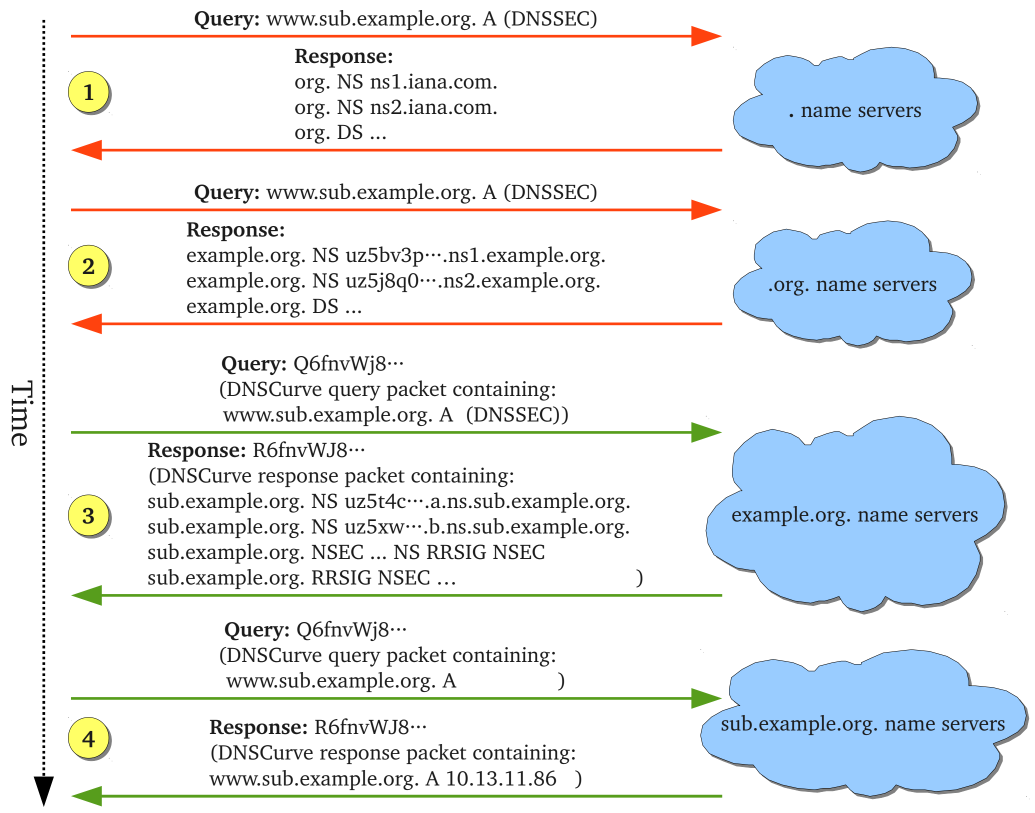
Time
Query: www.sub.example.org. A (DNSSEC)
Response:
org. NS ns1.iana.com.
org. NS ns2.iana.com.
org. DS ...
1
. name servers
Query: www.sub.example.org. A (DNSSEC)
Response:
example.org. NS uz5bv3p···.ns1.example.org.
example.org. NS uz5j8q0···.ns2.example.org.
example.org. DS ...
2
.org. name servers
Query: Q6fnvWj8···
(DNSCurve query packet containing:
www.sub.example.org. A (DNSSEC))
Response: R6fnvWJ8···
(DNSCurve response packet containing:
sub.example.org. NS uz5t4c···.a.ns.sub.example.org.
sub.example.org. NS uz5xw···.b.ns.sub.example.org.
sub.example.org. NSEC ... NS RRSIG NSEC
sub.example.org. RRSIG NSEC ...)
3
example.org. name servers
Query: Q6fnvWj8···
(DNSCurve query packet containing:
www.sub.example.org. A)
Response: R6fnvWJ8···
(DNSCurve response packet containing:
www.sub.example.org. A 10.13.11.86)
4
sub.example.org. name servers

Figure 4.9: *Graphical representation of a combined DNSSEC and DNSCurve traversal*

When looking at the traversal, it appears that in iterations 1, 2, and 3 DNSSEC is used, while DNSCurve is used in iterations 3 and 4. The traversal starts with a DNSSEC enabled query to the root servers, to resolve the address of `www.sub.example.org`. The name servers respond a referral (NS-type records), including a DS-type record and a corresponding signature, indicating the referral name servers support DNSSEC too. Note that no signature is included for the referral, because the fact DNSSEC does not sign non-authoritative data.

After successful verification of the DS-type signature (with a trusted local key), the `.org` name servers are contacted in iteration 2. They must support DNSSEC, according to the DS-type delegation, so a DNSSEC enabled query is send. The response looks like the one of iteration 1, only now with name servers for the `example.org` zone. Although this time the name server referrals consist of DNSCurve public keys.

The DS-type record will first be verified using a DNSKEY-type record of the `.org` zone that has not been shown. If this turns out okay, iteration 3 will start with a DNSSEC *and* DNSCurve enabled query for the `example.org` name servers.

The DNSCurve response that is received contains many records. There is still no exact answer for the query, but again a referral. However, it appears these name servers do not support DNSSEC. This is known because no DS-type record is included, this is confirmed by the denial of existence record and its corresponding signature, that tell there are only NS-, RRSIG-, and NSEC-type records for the `sub.example.org` response name. The name server names does however indicate `sub.example.org`'s name servers are DNSCurve capable.

When the verification of the denial of existence record turned out okay, the `sub.example.org` name servers are contacted in iteration 4, using DNSCurve and the same query as before. However, since the name servers do not support or provide DNSSEC, no DNSSEC responses are requested. The DNSCurve response contains finally an exact answer, finishing the traversal.

If the `example.org` name servers would not support DNSSEC, the response of iteration 2 would be vulnerable to active cache poisoning attacks. This can happen because the NS-type records are not accompanied with a signature and no delegation signer is sent along, that indicates the next zone is DNSSEC enabled. Therefore the authenticated chain would already end at the `.org` name servers.

This indicates a minimal DNSSSEC and DNSCurve approach that guarantees a fully secured traversal should support DNSSEC up to and including the first DNSCurve enabled zone, seen from the top.

4.7.2 Merging of DNSSEC and DNSCurve

In the previous subsection a minimal combined DNSSEC and DNSCurve approach was presented. It did however not say anything about the fact whether this would even work, nor did it evaluate any disadvantages. That is what will be discussed in this subsection, together with an approach that entirely merges DNSSEC and DNSCurve.

Looking back at the DNSSEC and DNSCurve combined traversal of Figure 4.9, it appears that both extensions can be used jointly. Iteration 3 in the figure uses both DNSSEC and DNSCurve for the query as well as for the response. This query will be accepted by a DNSCurve enabled forwarding or authoritative name server. The forwarder will simply pass the DNSSEC query along to a DNSSEC capable authoritative name server after decrypting. The authoritative name server will decrypt the query and then process it to give a DNSSEC response.

This is possible because benefit is made of the fact that DNSCurve is a link level protocol. It does not interfere with the DNS protocol's semantics, meaning that it does not matter what kind of packets it sends or receives, as long as these are valid DNS packets, because DNSSEC is backwards compatible, DNSSEC packets can be send and received by DNSCurve capable hosts.

However, the only benefits a user would gain of this combined approach is a higher security level, due to the new cryptographic primitives, and the fact the query name would be confidential (only when the streamlined packet is used).

This approach has a downside, in contrary of the security benefits. A client should now execute two cryptographic operations when a response is received. One to unbox the cryptobox and another operation to verify all DNSSEC related signatures. Besides, when it sends a query, it should perform another cryptographic operation to encrypt the cryptobox. This will affect the performance of a client. A data manager does now indirectly make a trade-off decision between security and usability for all the clients of the name servers he manages.

Let's focus on the traversal of Figure 4.9 again. If a client does not support DNSCurve, this means that the referral response of iteration 3 and the entire response of iteration 4 would be vulnerable to active cache poisoning attacks. Disproving the security guarantee that the entire traversal was secured. (Although, only benefit of a non-DNSCurve capable client in a combined approach scenario is that no 'double' cryptographic operations are done.)

The only way to solve this behavior is by enabling DNSSEC at all zones of a traversal. In this way the entire traversal will consist of a combination of DNSSEC and DNSCurve, calling it a 'merged' solution. Note however that DNSCurve will still not be supported by the root servers and TLD name servers. If this approach was applied to the situation of Figure 4.9, also the `sub.example.org` zone should have been DNSSEC capable. Now an entire traversal to resolve `www.sub.example.org` would be secure, even if a non-DNSCurve capable client is used.

Chapter 5

Curving in Practice

The regular DNS protocol together with its two security extensions have been introduced in theory, it is therefore time to focus on the practical side. Because this thesis will focus on DNSCurve, this chapter discusses the practicalities when deploying DNSCurve in an already existing DNS environment.

Next, since non-prototype DNSCurve implementations existed at the start of this master project, the development of a DNSCurve capable forwarding name server was also part of the master project. Hence, this chapter will also discuss the development of that name server, including a discussion of performance testing that was executed.

This chapter starts with a small introduction that describes the people involved in the creation of early DNSCurve prototypes. Next, a section is dedicated to the development of the forwarding name server, design choices and used techniques will be explained in this section. The section afterwards discusses the deployment of DNSCurve. While the final two sections will discuss the performance of the developed forwarding name server. The performance is measured using several aspects and compared with non-DNSCurve implementations.

5.1 Introduction

When DNSCurve was designed, the cryptographic primitives were already available via the NaCl project [7]. There was however no prototype that illustrated an implementation of the specification at that time.

The first prototypes that showed some of the implementation details of DNSCurve appeared at the end of 2008. Adam Langley [41] (employed by Google at that time) wrote the first publicly available DNSCurve related code material. This material consists of a Python[1] implementation of the

[1]Python is a heavily featured high-level programming language.

NaCl primitives, a Python tool set that is able to send DNSCurve queries towards DNSCurve capable name servers, and a primitive forwarding name server in C.

This code was adopted by Matthew Demspky [31] (employed by OpenDNS at that time) at the beginning of 2009. He started improving the code, and also altered it to comply with a new DNSCurve specification. Besides, he also wrote the earlier discussed draft RFC [30] for DNSCurve.

This forwarder prototype will act as a starting point for the DNSCurve implementation that is about to be discussed in the next section.

5.2 Implementation

As the introduction mentioned, not much code was available to demonstrate DNSCurve in practice. Therefore it was decided to develop a forwarding DNSCurve capable name server during this master project. This section will discuss the implementation phase, from design until testing.

The first subsection will explain the actual design choices. Next, technology that is about to be used will be discussed. Afterwards the actual implementation will be briefly explained. At the end of this section, the testing process will be discussed.

5.2.1 Design

Starting working at an implementation without a proper plan, or design, is asking for trouble. Therefore, deliberate design choices have to be made before the actual implementation can start. This subsection will enlighten the design choices that have been made to implement a forwarding DNSCurve capable name server, starting by discussing the requirements.

Before discussing these requirements, it is good to first make a general statement regarding the functionality of the forwarding name server:

> *The DNSCurve aware forwarding name server should be able to secure any authoritative name server with DNSCurve, in a correct, reliable, and fast matter.*

'Any' meaning every existing and widely used authoritative name server, while 'correct' implies that both DNS and DNSCurve conversations should work correctly according to its specifications. With 'reliable' it is meant that availability of the new forwarding name server should be as good as the regular DNS setup was, and by 'fast' meaning the forwarding name server should not give any noticeable speed degradation.

Yes, these notions are rather vague, but give nevertheless a good general impression what the forwarding name server should fulfill.

Requirements

The introduction section of this chapter mentioned there existed some DNSCurve forwarding name server implementations already. These implementations were however relatively simple and primitive, leaving much room for improvements. As mentioned at the end of the DNSCurve chapter, support for DNSCurve is small. However, there is a mailing list available for a select group of people interested in DNSCurve. The requirements discussed here, are a result of a small survey among members of this mailing list.

The introduction of this subsection gave a general (non-technical) definition of a forwarding name server. The requirements that are about to be discussed are of a more technical nature. Underneath each of these requirements a small description is given that explains what the requirement exactly means.

1. **Forwarding of regular (non-protected) DNS packets**
   This is a trivial requirement. Normal DNS queries must be forwarded to an existing name server.

2. **Detection of malformed regular DNS packets**
   Packets that arrive at the forwarding server that are not correct, should be discarded. Note this also includes malformed regular DNS packets, inside a DNSCurve packet.

3. **Detection of malformed DNSCurve packets**
   DNSCurve packets can also be malformed, in the sense that the forwarding name server was unable to open the cryptographic box correctly. This could for example be caused by accidental- or intentional bit corruption. Or by using a wrong cipher suite, including the wrong nonces, using the wrong server public key, etcetera. If this is the case, the forwarding name server should not respond to the query.

4. **Support for both DNSCurve's streamlined- and TXT-format**
   As can be read in the previous chapter, DNSCurve specifies two different formats. The first format focuses on not including redundant data, while the latter format focuses on backwards compatibility and is therefore more likely to pass through existing firewalls. The forwarder will support both formats, which is also required according to the DNSCurve specification [6].

5. **Caching of shared secrets**
   This is one of the biggest challenges for the forwarder implementation. Caching the shared secrets (i.e. the scalar multiplication of the client's public key and the server's private key) should give a significant speed improvement if the same client returns after a first transaction. Or, to be more specific, when a client returns with the same client public key.

What caching algorithm and settings should be used, is subject to research during implementation in accordance with performance tests.

6. **UDP and TCP support**
   When DNS is used over UDP, it has a packet limit of 512 bytes. If a packet is larger than this limit, the truncation (`TC`) bit must be set. In this case the client is recommended to send the query again, although this time over TCP.

   Since the addition of DNSCurve increases the size of DNS packets, though only constantly, support for TCP is a requirement too.

7. **IPv4 and IPv6 support**
   With an eye on deployment, it is good to also support the next generation data link layer: IPv6 [29]. When the forwarder logic is independent of the used data link and/or transport protocol, support for IPv6 can be relatively easy to implement.

8. **Nonce separation**
   The DNSCurve specification states that multiple name servers can share one key pair. This feature is interesting for servers that share the same IP address, but are physically located at different locations. This phenomenon is known as anycasting and heavily used by the root servers and TLD name servers. In this way high-availability can be achieved, together with enormous speed improvements. I.e. a host from The Netherlands will implicitly contact a root server located in Amsterdam, instead of one in Palo Alto.

   However, all servers in such anycast pool should be using the same key pair. To prevent any possible repetition (because the same shared secret will be used by a client while connecting to any of the servers in the pool), a nonce was built in. Still, it is theoretically possible two servers in this pool use the same nonce, for the same client. To prevent these nonces from colliding, a nonce separation mechanism should be put into place.

   This is done by selecting a fixed prefix per nonce, per server. If there are for example sixteen different servers in the pool, the first four bits of the 96-bits server nonce should be made fixed and different per server.

9. **Domain separation**
   Not all organizations have the ability to facilitate a new server for each existing name server. To make deployment easier, there should be support to forward queries to not only one existing name server, but to multiple. In this case forwarding depends on the query name, instead of on one fixed IP address. This would however also imply

the forwarding name server should look inside the DNS packet itself, instead of simply passing it along, which could influence performance.

10. **Multiple key pair support**
    There are organizations that supply a DNS service to other parties, but in such matter this supplying organization is not known to the party's clients. However, they use the same physical name server (and thus IP address) like the organization's.

    This is done by letting domain names of the organization being served by `uz5xg···.ns1.example.org`, while domains of the party are served by `uz5y3···.a.ns.example.com`. Both hostnames do however point to the same name server: `10.13.11.86`. This means that a physical server has to support multiple key pairs.

    However, the cryptographic library is not capable of detecting what public key was used by a client given a set of used key pairs yet.

Now that all requirements have been specified, it is good to give each of them a priority. This is done by using the MoSCoW-method, that categorizes each requirement in a certain priority class. Ranging from 'must have', 'should have', and 'could have' to 'won't have'. Table 5.1 portrays the requirements ordered by their respective priority.

	Requirement	**Priority**
1	Forwarding of regular (non-protected) DNS packets	Must have
2	Detection of malformed regular DNS packets	Must have
3	Detection of malformed DNSCurve packets	Must have
4	Support for both DNSCurve's streamlined- and TXT-format	Must have
5	Caching of shared secrets	Must have
6	UDP and TCP support	Must have
7	IPv4 and IPv6 support	Should have
8	Nonce separation	Could have
9	Domain separation	Could have
10	Multiple key pair support	Could have

Table 5.1: *Forwarder requirements ordered by their respective priority*

The priority is based on several factors. Besides trivial requirements, the most important one is time. Although, also complexity and popularity have been considered. When the actual implementation is finished, this table will come back, for a small evaluation.

From these requirements several use cases have been created. These use cases are in fact scenarios exploiting all distinguishable situations sketched by the requirements. These use cases are not only convenient during implementation, but also when the implementation will be tested. Because they describe almost all possible scenarios the forwarder can come across when being deployed in the wild.

5.2.2 Technology

Design choices are not only focused on functionality, technology that will be used is just as important. This subsection will discuss technology related choices. From general choices (such as the used programming language) to more specific ones (i.e. what cryptography-, event-, and network stack should be used).

General Choices

As mentioned, some choices regarding the forwarding name server have already been made. This text will clarify what general choices have been made, and – even more important – why these choices have been made.

When looking at technology of software, almost the first thing that comes to mind is the programming language that is going to be used. Sometimes choosing a certain language directly implies the use of an associated platform. A platform entitles the whole set of options that facilitate everything that is needed to run a program. Among these options the operating system (e.g. Unix, Microsoft Windows, Mac OS X, etcetera), but also the system's CPU architecture (e.g. x86, AMD64, ARM, SPARC, etcetera) can be considered. However, since the introduction of high-level programming languages, platforms are given a lower priority in the decision for a language.

To be a bit less abstract, different popular authoritative name servers have been analyzed, to find out what programming language they use and what platform they run on. Although the forwarding name server that is about to be implemented is an entity on its own, a comparison with existing solutions might seem awkward. However, in general a data manager will easier switch to a forwarding solution that uses the same platform (and preferably language) like his current DNS setup uses.

A fairly recent DNS server survey [46] shows the four most used name servers: BIND, tinydns, followed by Microsoft's DNS server, and finally PowerDNS. BIND is written in C and can be run on almost any platform. tinydns is also written in C, it can however only be run on POSIX enabled platforms. It is unclear in what language Microsoft's DNS server is written[2], it can however only be run on Microsoft Windows. PowerDNS on the other hand is written in C++. There is the possibility to run PowerDNS on Microsoft Windows, this would however involve compiling your own packages, although regularly PowerDNS runs on POSIX platforms.

[2]It is however believed Microsoft's DNS server is written in either C or C++, because it is a very early fork of BIND.

Because three of the four most used authoritative name servers run on a POSIX platform, development of the forwarding name server will focus on this platform too. This choice implicitly covers support for many Unix-like operating systems, such as Mac OS X, FreeBSD, and also the popular GNU/Linux.

Almost all high-performance daemons that run on POSIX platforms are written in C. Examples are the Apache webserver, popular mail transfer agents such as Postfix, qmail, and Exim, and FTP daemons such as ProFTPd, vsftpd, and pure-ftpd. Why C is used in so many applications is caused by several factors.

The most important one is the fact that most of the POSIX operating systems are written in C themselves. This means almost all of these systems have a C compiler toolkit delivered with them – or at least the ability to obtain one. Another advantage is that making API calls (or: syscalls) to the operating system will be easier in this way. Something else that influenced the choice for C, is the fact that most of the libraries that are about to be discussed are all written in C as well. This will simplify interfacing with these libraries. Another reason why C is used by so many daemons, is the huge amount of control a programmer has over the program's memory. Besides, C programs run directly on the operating system, without for example a VM in between, making them faster than VM-based languages.

Disadvantages of C are indirectly caused by its advantages. Because of the large amount of control a programmer gets, mistakes are made easily. The best known mistake is the so called 'buffer overflows'. These can lead to major security problems. They exist because C does no runtime type and boundary checking on any of its types.

This means that programmers programming in C should take care with all data they handle in their program. Therefore, secure programming principles should be used, like using safer variants of functions, such as `strncpy` instead of `strcpy`. Dynamic memory should first be zeroed, before it is used. Also, extra care should be taken when data in dynamic arrays is being used.

Cryptography Stack

Both the DNSSEC and the DNSCurve chapter dedicated a subsection on cryptography. Now it is time to implement the cryptographic primitives explained in the DNSCurve chapter. Since there are no other libraries that support the newly defined primitives, it is straightforward to see why the NaCl library [7] will be used.

Let's take a look at the specific primitives that are about to be used in the DNSCurve forwarding name server. As mentioned in chapter 4, DNSCurve makes use of three combined cryptographic primitives to deliver a cryptographic system that provides both confidentiality and authenticity. Or, to be more precise and use the terminology of NaCl, privacy and third-party unforgeability. These combined primitives are the elliptic curve Diffie-Hellman key exchange CURVE25519, the stream cipher XSALSA20, and finally the message authentication code POLY1305. This combination is bound together in NaCl's 'public-key authenticated encryption' API, and is referred to as the 'cryptobox'. Notice however that the CURVE25519, XSALSA20, and POLY1305 is not the only cryptobox primitive that is supported by NaCl. Also the NIST P256 curve, the AES-256 block cipher, and the HMAC-SHA-512 MAC combination qualifies as a cryptobox.

Due to the modular setup of NaCl, there should be a way to call all different individual primitives or a combination of them. To facilitate this, NaCl uses a standard naming convention and C pre-processor definitions. So to use the different cryptobox combination, a programmer can call the `crypto_box_nistp256aes256hmacsha512` function. DNSCurve on the other hand will use the `crypto_box_curve25519xsalsa20poly1305` function.

Besides different primitives or combinations, each cryptographic primitive or a combination thereof can also have different implementations. For example, the CURVE25519 primitive has three different implementations. One general implementation, but also an AMD Athlon CPU specific implementation. The NaCl build system will automatically select the appropriate implementation, i.e. the fastest functioning one.

Back to the cryptobox primitive. The `crypto_box` API comes with three functions for different purposes: a key generation interface, an encryption plus authentication interface, and a decryption plus authentication verification interface. The latter two functions have also a split functionality, meaning the shared secret generation can be separated from the actual boxing process. Due to this intermediate step, the shared secret can be cached.

With all these functions, everything is there to fulfill DNSCurve's cryptographic needs. Because the library choice was already (implicitly) made, there was no room for discussion. Therefore this text focused on the library itself.

Event and Network Stack

This part will discuss what event handling solution has been chosen, together with the used networking stack.

Event handling is of great concern in daemon applications. In short, any action inside a system can be considered an event. For example, if something is received from the network, when a timer expires or when a line is ready to transmit data, are all actions that result in an event. To detect these events, several solutions exist.

The least efficient solution is to constantly keep checking whether there is an event. This is called busy-waiting and wastes a lot of CPU time, because the system will repeatedly check the condition that in many cases did not change in the meanwhile. This makes the CPU less available for other duties and this solution should therefore be avoided.

Operating systems supply more efficient solutions. One of the oldest solutions is to block until an event is received. The system's kernel facilitates the event handling. This is the standard solution on nowadays Unix implementations. When for example a request is made to sleep a process for the next 60 seconds, the call will block, i.e. it will not do anything until the event has actually been generated, meaning that also no CPU time is wasted. Note however, that when the call is blocked a program cannot execute any other instructions in the meanwhile.

Therefore, also a non-blocking solution is supplied. A call that is non-blocking will directly return with an error when there was no event and if there was event, it will process like it would regularly do. This solution looks like busy-waiting, since a process would constantly have to check whether an event has been received or not. However, operating systems supply a way to handle this efficiently: the process itself will be notified by the kernel if an event is received.

Another approach is that a process can inform the operating system's kernel it would like to receive these notifications for particular events. The oldest solution that provides this is the `select(2)` call. The program will supply a list of events (i.e. so called file descriptors) to `select(2)` that it would like to be informed of when they occur. `select(2)` will now block, until an event has been received, or when a timeout value has been reached. The biggest advantage of this solution is that the process can wait for multiple events by using one single blocking call.

Besides `select(2)`, several other similar solutions are developed, such as `poll(2)`, `kqueue(2)`, `/dev/poll`, and `epoll(4)`. Functional differences between these solutions are relatively small, although when looking at performance numbers vary, especially when the number of events that have to be watched grow [38].

Disadvantage of all these different solutions is that not every solution is available at every POSIX-capable platform. For example `epoll(4)` is solely available under Linux, while usage of `/dev/poll` is limited to Solaris. Therefore, it was chosen to use a library that will use the appropriate mechanism that is available on a system.

Two generally used libraries are available for this purpose: libevent [56] and libev [43]. Benchmarks show that libev outperforms libevent's stable release in the average case, especially when many events are being watched [43]. Besides, libev includes good documentation and there is community support available. It is also able to watch for timeouts and signals simultaneously. This greatly improves the simplicity of implementing a way to handle both, therefore it was decided to use libev in the forwarding name server.

There is not much to say about the networking stack that will be used, since all POSIX enabled platforms support the so called Berkeley Socket API, it is clear this API will be used. Because this API is used so much, good documentation and examples are available online as well as in literature [28] [58].

Miscellaneous

One of the requirements that has to be implemented, is to use a caching algorithm for the shared secrets that have been generated. It is known from the cryptography section that CURVE25519 keys, both the public and private key, are all 32 bytes in length. This caching algorithm should provide a way to lookup the 32-byte shared secret, given a 32-byte public key. The corresponding data structure should be able to store a collection of values, where each value is associated with a unique key. If no shared secret is found, one should be computed and inserted into the data structure afterwards.

There are many algorithms known that, given the data structure, fulfill this purpose. The simplest would be maintaining a list of all public keys and associated shared secrets. The lookup algorithm would check all public keys, resulting in a performance of $\mathcal{O}(n)$ for lookups and $\mathcal{O}(1)$ for insertions.

An improvement in lookups can be achieved by sorting the public keys lexicographically. Many different implementations exist for this approach that all guarantee that lookup, but also insertion, can be done in $\mathcal{O}(\log n)$ time. This however degrades insertion performance in comparison with the initial approach, while insertion is just as important. Therefore a data structure and associated algorithm is needed that is able to execute both operations in preferably constant time.

Due to time limitation, only one data structure will be considered: hash tables. Depending on what type of implementation is used for the hash table, in general a hash table provides insertion and lookups that can be done in $\mathcal{O}(1)$ time. Hash tables have a finite number of so called 'hash buckets' available. During a lookup, the search string is hashed (by using a hash function – not to be confused with a cryptographic hash function) to be related to a certain bucket. It is then checked whether the key at this bucket equals the search string, if this is the case the value (i.e. the shared secret) will be returned. If they do not match, it depends on the implementation what will happen, although the forwarder's implementation uses name chaining. Meaning the bucket is the start of a linked list that contains all keys that have the same hashed value. In worst case, this makes the hash table $\mathcal{O}(n)$ (i.e. all keys hash to the same bucket), although when a good hash function is used, the average performance will still be $\mathcal{O}(1)$.

Note that the algorithm will be slightly adapted, because only a finite amount of shared secrets can be stored, a refreshment system is needed. This system will drop the oldest stored shared secret when the maximum amount is reached, and fill it with the newly added pair. This involves extra memory usage because more administration is needed. A benchmark (conducted in subsection 5.5.3) will test the performance of the implementation.

5.2.3 Implementation

Now that design choices and the technology that will be used have been explained, it is time to focus on the actual implementation. This is done by explaining the flow of a DNS or DNSCurve conversation that is initiated by a client. The flow is portrayed by two illustrations. Note that this subsection will give a global explanation, many details have been left out or simplified.

Figure 5.1 shows what will happen when the forwarding name server receives a DNS or DNSCurve query, sent by a client.

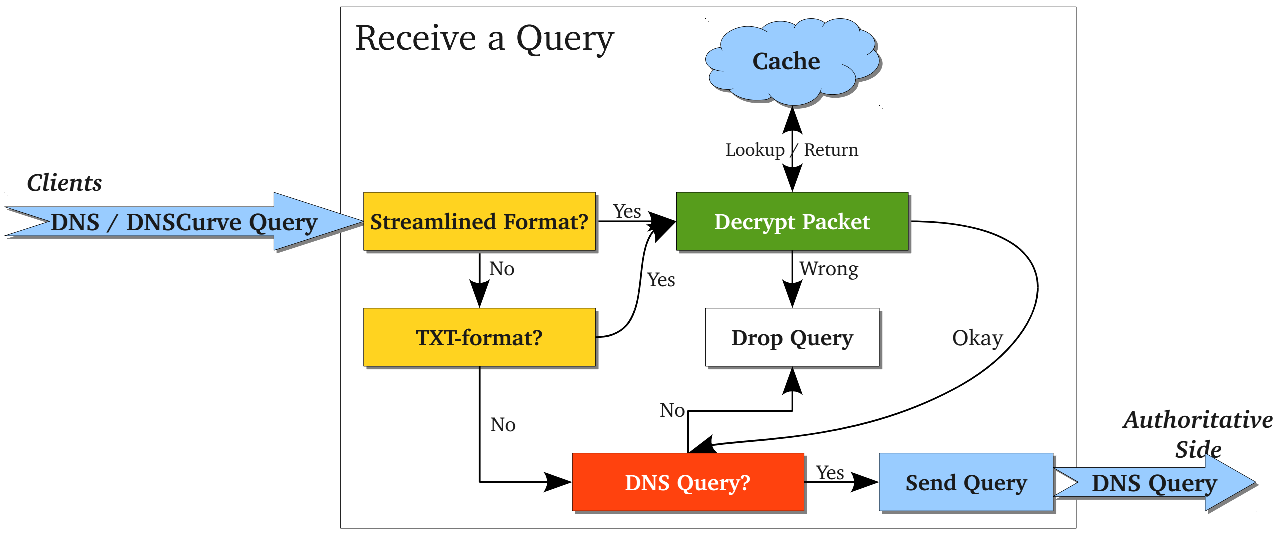

Figure 5.1: *Schematic flow of a DNS or DNSCurve query that is received from a client*

The forwarding name server is able to accept queries received over UDP or TCP. Besides both transport protocols, the forwarding name server also supports both IPv4 and IPv6.

When a query is received, it will directly check whether it is a DNSCurve streamlined query. This can be determined quickly, by checking whether the packet is prefixed by the magic string (`Q6fnvWj8`). If this is the case, the packet will be decrypted, when it is not a streamlined DNSCurve packet it will be checked whether it might be a DNSCurve TXT-format query. Checking this is relatively harder, it should be examined whether the query name contains the client's public key and if it is a TXT-type query. If it turns out to be a TXT-format query, the needed information is extracted and the cryptobox will be decrypted. When it is not a TXT-format query, it is considered to be a regular DNS query.

The decryption process starts by checking whether there already resides a shared secret for this particular public key in the cache. If it already was in the cache, this shared secret will be used. When this is not the case, a shared secret will be computed and directly added to the cache. Next, the cryptobox will be opened by using the shared secret and the client's nonce that was included in the query. If the opening fails the query will be dropped, on success the query is considered a regular DNS query.

In the red box it will be determined if it truly is a DNS query (i.e. whether all bit flags are okay and only one query and no answers are included). If this turns out wrong, the query will be dropped. When everything is okay, the query will be send towards the authoritative name server. Although, not before the identifier of the DNS packet is changed to a different random number. This is done to make it harder for an attacker to attack the authoritative side of the forwarder, more on this was explained in the previous chapter.

Note that some bookkeeping is done before the query is sent. A timer is set that will expire when no response is received. When the timer expires and no response has been received, it depends on what the data manager specified to respond. Besides the timer, some other information is saved that is valuable when the response has to be sent. Think of the client's address information (i.e. the source IP and port) and the client nonce plus shared secret, when it was a DNSCurve query.

Figure 5.2 shows what is done when a response is received from the authoritative side.

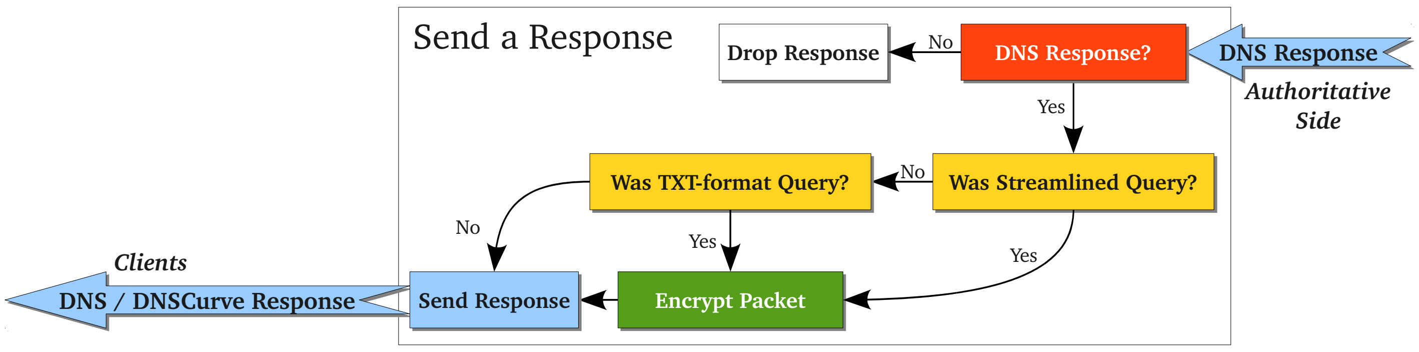

Figure 5.2: *Schematic flow of a DNS response that is received from the authoritative side*

First it will be checked whether the response truly is a DNS response. If this is not the case, the response will immediately be dropped. When it is a DNS response for the right query, the identifier will be replaced with the one of the query. Next, it will be checked whether the query that initiated this response was a streamlined DNSCurve query. If so, the packet will be encrypted, when it was not a streamlined query, maybe it was a DNSCurve

TXT-format query. If it was a TXT-format query, the packet will be encrypted, otherwise the response is ready to be send back to the client.

Note that when a packet needs to be encrypted, no use is made of the cache, because the shared secret was saved in the bookkeeping process before the query was sent towards the authoritative side. The encryption needs the actual response packet, the shared secret, the client nonce (also saved temporarily), and a server generated nonce. If the encryption step is done, the query can be send back to the client. This is done by using the same protocol (i.e. UDP or TCP) as the query arrived in. When the response has been sent, the internal bookkeeping for this specific query can be cleared.

This concludes the description of the implementation. The requirements from the design subsection can be seen in Table 5.2, together with the fact whether they are implemented or not.

	Requirement	Priority	Impl?
1	Forwarding of regular (non-protected) DNS packets	Must have	Yes
2	Detection of malformed regular DNS packets	Must have	Yes
3	Detection of malformed DNSCurve packets	Must have	Yes
4	Support for both DNSCurve's streamlined- and TXT-format	Must have	Yes
5	Caching of shared secrets	Must have	Yes
6	UDP and TCP support	Must have	Yes
7	IPv4 and IPv6 support	Should have	Yes
8	Nonce separation	Could have	Yes
9	Domain separation	Could have	No
10	Multiple key pair support	Could have	No

Table 5.2: *Implemented requirements of the forwarding name server*

Only the domain separation and multiple key pair support features have not been built in. Domain separation was left out because of time constraints, while multiple key pair support was not yet available in the NaCl library.

5.2.4 Testing

Testing functionality that is implemented in the forwarding name server is of course of great concern. Therefore, parts of the already developed DNSCurve Python tools [31] [41] will be used during development to test the name server's behavior. These tools are able to produce streamlined and TXT-format DNSCurve queries as well as regular DNS queries. Besides, they can interpret the responses, so that a nice summary of what is received can be given. The output can be compared with the output of the popular `dig(1)` tool, that represents a regular DNS response.

If these Python tools (and the `dig(1)` tool for regular queries) would not give an explainable answer when strange behavior is encountered, the Wireshark packet sniffer can be used to determine what exactly was transmitted over the communication line.

The earlier described use cases will also act as testing scenarios. Because they describe in detail almost every possible scenario the forwarder can encounter. Next, because OpenDNS' recursors are DNSCurve capable, a query to one of the OpenDNS caching name servers would involve their resolvers to contact the forwarding name server by using DNSCurve. In this way an independent test can be conducted.

When development is in a later stage, the address and public keys of the forwarder can be published to other people interested in DNSCurve. In this way, they are able to test their respective implementations against 'our' forwarder.

5.3 Deployment

Protocol deployment and adoption on the Internet has always been a difficult matter. Many examples support this claim, think of the worldwide adoption of IPv6 [29], that still does not have a priority. Also the adoption of DNSSEC suffered the same problem, although at this moment deployment is slowly starting to take place.

This section will discuss the speed of deployment in relation with DNSCurve. Furthermore, the actual process of deployment will be discussed in detail.

5.3.1 Speeding up Deployment

There are quite some reasons imaginable that caused (and causes) slow DNSSEC adoption. Firstly, because there was no root server support until July 15, 2010. This made DNSSEC relatively useless without using an external trust delegation system. Secondly, because users did not see any of the benefits in upgrading their DNS security, i.e. 'everything works now, why should we change'. Thirdly, the complexity of the DNSSEC protocol might back off novice data managers and users. Fourthly, a data manager stated he will only start deployment when related data managers are done deploying, which results in a never ending circle. Fifthly, because of the many changes during the DNSSEC development, data managers might lost confidence in the protocol. More on protocol adoption models – including a case study on DNSSEC – can be found in [52].

Let's return to the DNSCurve environment. To make deployment of DNSCurve easier, several things should be considered.

Users that are interested in something, want to have more and concise information regarding this matter. Linking this to DNSCurve, potential users are more likely to try DNSCurve when good documentation is available. Documentation on many different fields, e.g. a good description of what DNSCurve now exactly is, what it solves, and the literal meaning of documentation, i.e. a good explanation how to install a DNSCurve enabled forwarder for example. But also performance (white-)papers can persuade people to give DNSCurve a try.

When non-private users, such as (commercial) organizations are interested in adopting DNSCurve, other information is needed. Such organizations usually relate benefit to cost. Numbers regarding this matter would be of great help in convincing organizations to start deploying DNSCurve. For example, how much manpower it would take to create a DNSCurve capable replacement for an organization's entire DNS setup.

Come one, come all is a saying that can also be applied to protocol adoption. If great players in the market start to deploy a certain protocol, this will inherently get others to deploy the protocol too. Although there is no solution out there yet that supports DNSCurve at the authoritative side, OpenDNS announced at the beginning of 2010 their resolvers are DNSCurve capable [32]. OpenDNS can be called quite a big player when it comes to open DNS resolving. They take care of DNS resolving for several large enterprises and according to their system status page [50], OpenDNS handles about 23 billion DNS queries per day. This support will positively influence authoritative data managers that consider switching to DNSCurve.

5.3.2 Deployment Process

Initiating a DNSCurve capable environment is one thing, maintaining it is another. Note however there is usually a big difference between the client environment and a server environment.

Switching to DNSCurve in the client environment is relatively easy. The only thing that needs to be done, is to replace the current used resolving software at a caching name server for a DNSCurve capable equivalent. No public released source code is available that patches current resolver implementations to support DNSCurve yet. This would make the transition even easier, as administrators of these systems can still use their favorite software implementation.

Note however that only a caching name server that is DNSCurve capable at the client side, does not give an entire secured topology. DNS conversations between stub resolvers and the caching name server are still vulnerable to all regular DNS related attacks.

The server environment is a whole different story. In this environment a transition to being fully DNSCurve deployed takes more effort. The DNSCurve chapter already stated the steps that a data manager should take to initiate DNSCurve deployment. Although nothing was mentioned on the administrative side of this transition.

When a keypair is generated for an entire authoritative name server, the private part of this key should be kept secret at all times. Besides secrecy, also availability of the private key is of concern, i.e. if the private part is lost or not available, the server cannot encrypt and decrypt any of the cryptoboxes, making the keypair useless. Making a backup of the key material must therefore be considered mandatory.

But what if a key was compromised, lost, or a year has passed and the artificial lifetime of the key has expired? This all means a key rollover should take place. This process involves the generation of a new keypair, and the deployment of the new public key inside the name server's name. Notice there is a difference between a rollover because the key was compromised and a lifetime replacement. The first indicates an emergency rollover, since in the meantime an attacker is able to send rogue responses on the name server's behalf. Emergency rollovers are relatively hard to achieve in a short period of time. Since the server's public key (and inherently the namer server's domain name) can reside in a client's cache for the NS-type record's TTL amount of time. If a client with such old public key in its cache would contact the name server that just rolled over a new key, this query will be discarded. Nevertheless, the regular DNS availability mechanism will automatically try the zone's secondary name server. It is unlikely this name server is also unavailable for whatever reason. A coordinated (i.e. non-emergency) key rollover can be done by lowering the TTL of the NS-type records some time before the actual rollover takes place. So that a change of these records will be quickly noticed by a client.

This concludes all remarks regarding deployment of DNSCurve. The up following final two sections will discuss the performance of the forwarder.

5.4 Performance

Developing and implementing a forwarding name server implementation is one thing, finding out if the application suits one's performance needs is something else. Performance of the implementation will be tested by using specific benchmark tools. This section will explain how the forwarding name server performs in specific situations.

The first subsection will introduce the used benchmark tools. Next, the benchmark setup will be explained. The subsections that follow discuss the actual performance tests. Ranging from speed benchmark comparisons to specific shared secret caching algorithm performance tests.

5.4.1 Benchmark Tools

When looking at specific DNS benchmark tools, it appears that most toolkits focus on testing caching name servers [4] [59]. On the other hand, NIST has a performance testing tool [5] available that focuses on testing authoritative name servers. NIST built this tool to be able to test the performance of DNSSEC capable name servers. Besides, Nominum Inc. (a DNS software developer) also developed a DNS query performance application, called dnsperf [48]. None of these tools do however currently support DNSCurve. Therefore it was decided to build a new benchmark tool that supports DNSCurve and focuses on performance testing of authoritative name servers. Nevertheless, the dnsperf tool will still be used, to test the forwarder's performance on regular DNS queries.

This new tool will be heavily inspired by the way the NIST tool works. The NIST tool consists of sample data and several separate tools, each with an individual purpose. To performance test a specific server, first a traffic model has to be made. This traffic model describes – in percentages – what type (i.e. A-, MX-, or NS-type) of queries should be made. Besides what type of queries, the model also describes what portions of queries are intentionally wrong. Examples hereof are non-existent query names or for example queries containing semantic errors.

When a model is ready, a traffic generator tool will take this model file as input together with the zone data the authoritative name server serves. The NIST tool consists of anonymized zone data, that originated from real zone information. The traffic generator will generate traffic according to the model and the zone data file and save it to a different output file.

Another tool will load the traffic file and start querying the queries from this output file towards the authoritative name server that is about to be tested. This tool reports back the total number of queries that have been sent at a particular interval in time. So it knows the query rate (measured in queries per second), success rate (how many of the queries got an actual response), and also average bandwidth rate (measured in bytes/second).

In the introduction of this chapter, it was already mentioned DNSCurve prototype implementations were written in Python. Because of the ease of development in Python and the already available code regarding DNSCurve in this matter, the benchmark toolset will also be written in Python.

Traffic Generator

The model format used by NIST is a good inspiration, although not all options it features are useful in our benchmark. Therefore the format is simplified, while DNSCurve support is added. Figure 5.3 illustrates an example model file that is compatible with the developed traffic generator.
Note that all lines starting with a hash (`#`) are comments. The first interesting line is the one starting with `settings`. This line specifies the settings of the traffic load file that is about to be generated. The first number tells how long a benchmark should be able to run (in minutes) for a given number

```
# settings: #MinsToRun #qps
settings: 10 100

# dnscurve: #DiffKeysToUse PublicKeyServerInBase32
dnscurve: 100 uz5dk3v...

# type %ofTotal %nameErr %DNSCurve
1 0.70 0.15 0.60
2 0.06 0.05 0.30
5 0.08 0.10 0.50
6 0.02 0.10 0.40
15 0.12 0.05 0.80
16 0.02 0.10 0.20
```

Figure 5.3: *Example DNSCurve benchmark model file*

of queries per second (qps), i.e. the second number. Notice this does not tell how long the benchmark will actually run, it only indicates the traffic generator will generate a traffic load file that is able to run for such long period. (In this case it will create a file that contains 60,000 queries.)

What follows is the `dnscurve` line, that specifies all DNSCurve related settings. If it is left out, no DNSCurve queries will be made. The first number indicates how many different DNSCurve client keys should be used in all the queries. This is an interesting number when a DNSCurve server's shared secret cache will be benchmarked. The second field is the base-32 encoded public key of the name server, prefixed with the magic `uz5` string.

Next, the record type specific lines follow. These lines all have the same format, distinguishing four different values separated by spaces. The first value indicates the record type number [9]. The lines represent A-, NS-, CNAME-, SOA-, MX-, and TXT-type records respectively. The second field specifies what percentage of the total number of queries should be of this query type. In the example, A-type queries involve 70% of all queries, while 2% of the queries ask for TXT-type records.

The next field indicates what percentage of the queries for this particular record type should have non-existing query names. In the example 15% of the A-type queries should return a `NXDOMAIN` response, because the query name does not exist.

The fourth field shows what percentage of the record specific queries should use DNSCurve. 60% of the A-type queries will be secured by DNSCurve. Note that both the name error and DNSCurve portion percentages can range between 0 and 100%, i.e. they are independent of the second field's percentage.

This model specifies that a generated traffic load file will consist of nearly[3] 40,000 A-type queries. 6,300 of these queries will have a non-existing query name, and 24,000 of these queries will be protected by DNSCurve. Notice that both can be combined too, i.e. a DNSCurve query can also contain a wrong query name.

The traffic generator that is developed in Python is called ‘dnsprepare’. It needs two different input files: the model, and a data file that contains all zone information. This data file should be in tinydns’ data format[4].

dnsprepare will pre-compute all DNSCurve shared secrets and also save them in the output file – that will be discussed later. Furthermore, it will generate the entire query packet and base-64 encode it before the query is saved. In this way, the traffic benchmark tool will only have to base-64 decode the string, and it can directly send it towards the name server. Also DNSCurve queries are pre-computed in this way, avoiding intensive CPU tasks during the actual benchmark.

Traffic Benchmark

Now that there is a traffic load file available – that contains queries according to the specific traffic model – it is time to actually do something with these queries. The input file of the traffic benchmark tool, that is called ‘dnsbenchmark’, is the output file of the dnsprepare tool. The format is portrayed in Figure 5.4.

The settings line is a one-on-one copy from the model file, and indicates the number of minutes to run the benchmark followed by the queries per second (qps) to fire at the authoritative name server, respectively.

What follows are the shared secrets, prefixed with the number of the secret. Note that all secrets are base-32 encoded, however a different base-32 alphabet than DNSCurve uses. This file contained 100 different shared secrets.

Afterwards, the query lines can be found. These fields inside these lines are again separated by a space. The first number indicates the query type. The following number tells whether it is a non-existing query name (value 1) or not (value 0). The third number indicates if the query is a DNSCurve (value 1) query or not (value 0). What follows is the actual query name. Consider the third line of the queries for example. This is an A-type query, with a non-existing query name, it is also a DNSCurve query. Although all query names look relatively random due to NIST’s anonymizing step, the string ‘xxzz’ can be distinguished. This extra string will make the query name non-existing.

[3]Nearly, because the generated traffic file is based on random values that are not perfectly distributed, leaving some room for ‘errors’.

[4]tinydns is part of the already introduced djbdns package [17] in the DNSCurve chapter. It is an authoritative name server written by Daniel J. Bernstein that is fast and features a well structured input data format.

```
# generated by dnsprepare traffic generator on: 2010-07-19 17:56:41
settings: 10 100
dnscurve-shared-key: 0 xPA/90sbGb7JsvWZoYG2QMWFN8p3FNLjaMWLo2lCMN4=
            .     .     .     .     .     .     .     .
dnscurve-shared-key: 99 8BAyyhHXiAQlX+9k3NGSY8qJC1iXJKgSSVrgUESTZx8=

2 0 0 lidoad.if Fr4AAAABAAAAAAAABmxpZGeizeyDwKTbh8CLvv3XjS...
2 0 1 fnkivicy.if 72 UTZmbnZXajggxxe8yMQVmcCzsjWxRTbhZagRl...
1 1 1 bjweh2xxzz.fraul.vwr 13 UTZmbnZXajjyVizyDwKxKMjANeCh...
5 0 1 fff.hpn.fraul.vwr 65 UTZmbnZXajgETeh8F+90/jEZLVfEde/...
1 0 0 tvr137.fraul.vwr 3UMAAAABAAAAAAAABnR2cjEzNwVmcF1bAV4...
1 0 0 huaXwtaXusn.fraul.vwr 5q8AAAABAAAAAAAAC2h1YVh3GFYdXN...
1 0 0 auehaXfjt.fraul.vwr XssAAAABAAAAAAAACWF1ZWhhWGqdAVmc...
1 0 0 nvlv01.fraul.vwr ce4AAAABAAAAAAAABm52bHYwMQVmcF1bAXt...
15 0 1 vjtauvjt.fraul.vwr 33 UTZmbnZXajjWxRTbh8CLvN31Qw2D3...
5 0 0 nkes.fraul.vwr BvYAAAABAAAAAAAABG5rZXMFZnJhdnXaS8YjT...
1 0 1 uveqojtdqvtuvh3.fraul.vwr 61 UTZmbnZXajjxrWSFvtI4s+R...
            .     .     .     .     .     .     .     .
```

Figure 5.4: *Example DNSCurve benchmark traffic load file*

The format following after the first four fields depends on whether the query uses DNSCurve or not. If it is a regular query, the fifth field represents the base-64 encoding of the query packet.

If the query is a DNSCurve query, the fifth field is the shared secret key number from the top of the file that was used to generate the query. This shared secret is needed in order to be able to decrypt received responses. The last field is again the base-64 encoding of the DNSCurve query packet.

The dnsbenchmark can be run with several parameters. The traffic file together with the target IP and port of the server are mandatory. Furthermore, the tool can be called with a different number of minutes to run, also the qps parameter can be changed. A negative number of queries per second will let the tool run forever. If not enough queries are supplied by the traffic file, the tool will reuse the same queries again. Note this tool is only able to send queries to one specific host at a time.

The specified number of queries per second is a guideline. If the server – or the client – cannot keep up with this limit, the tool will report this when it is running, i.e. it shows the actual achieved qps.

Besides being a high-performance benchmark, dnsbenchmark can also actively look inside responses. In this way validating the functional behavior of an authoritative name server can be tested. It can for example look whether it truly received `NXDOMAIN` responses and if the packet identifiers match. Note however this functionality is turned off by default, especially for DNSCurve responses. Because this would cause a decreasing qps performance, since the client has to unbox all DNSCurve responses. This default behavior can however be altered by supplying a specific ‘response inspection’ parameter.

When dnsbenchmark is running, it will report at certain intervals the number of queries it sent, received, or that timed out. If packet inspection is turned on, it will also state how many queries resulted in an actual answer, a negative response (i.e. `NXDOMAIN`), or a wrong response (wrong bit flags, no corresponding identifier, etcetera). Eventually, a simple straightforward summary is given per category (regular DNS and DNSCurve) telling how many queries went okay or not.

5.4.2 Benchmark Setup

Benchmarking tools are one thing, a platform to run on is something else. This subsection will describe the setup, hard- and software wise, that is used to perform all benchmarks on.

Hardware

All systems used in the benchmark are physically located on the same machine. This machine has two Intel Xeon 5130 (4MB L2 cache) CPUs on board, that both operate at 2GHz. Note that both CPUs consist of two cores, meaning that four cores are available. In total the machine has the ability to use 9.0GB of RAM, indirectly implying the CPUs are 64-bit ready.

As mentioned, more systems are virtually allocated on this machine, so that virtualization is used. The system operates on VMware ESXi version 4.0.0.

Now that the physical machine itself is introduced, let’s focus on the so called guest hosts. Three different guest hosts have been distinguished, each with a separate task in the benchmark. This division will be explained later. Although their function is different, under the hood everything is the same. Each system has two (virtual) CPUs it can use, 512MB of RAM is available and the systems are interconnected by a virtual Ethernet connection.

Note that the system houses more guest hosts that are not relevant for this thesis. All of these guest hosts did not use any resources when the benchmarks were performed.

Software

Looking at software, all systems run 64-bits Debian GNU/Linux Lenny version 5.0.5 and operate on Linux kernel 2.6.26-2-amd64 (i.e. a Debian altered version). Furthermore, every system has the following software packages installed that matter for this thesis’ benchmarks:

GCC The GCC C compiler (GNU Compiler Collection) is used to compile NaCl, tinydns, and the forwarding name server. All systems have version 4.3.2 (Debian 4.3.2-1.1) installed.

Python The Python interpreter is used to be able to run the benchmark tools, that are written in Python. Python version 2.5.2 is used.

libev The libev library [43] was explained in more detail in the technology subsection of the implementation section of this chapter. It facilitates a standard way in handling events coming from various sources. At all machines version 3.43 is installed. libev can use different event handlers, during the benchmarks the `epoll(4)` event handler will be used by libev – note this is the standard and best performing choice when Linux is used.

NaCl NaCl [7] should be familiar to the reader by now, it has been introduced and explained in the DNSCurve chapter, but also in some detail in this chapter. All systems have the version installed that can be found inside the `nacl-20090405.tar.bz2` package.

This is the standard software that is installed at each server. Furthermore, also Bernstein's daemontools package is installed. This package makes management and control of daemons easier. As mentioned, three different systems have been distinguished:

1. **Authoritative Name Server**
   This machine will host the authoritative name server. From the benchmark tool subsection, it could already be read that tinydns [17] will be used as authoritative name server. However, the standard tinydns 1.05 version has been modified with two minor patches.

   The first patch[5] changes tinydns' behavior in handling non-existing responses. Normally, tinydns will not respond to queries it is not authoritative for, i.e. it does not have an answer for. However, during a benchmark a client should not wait for a timeout to conclude the server is not authoritative for this query. Therefore, tinydns is altered in such a way it will respond with the `RCODE` set to `NXDOMAIN` when it has no answer available.

   The second patch[6] modifies the way tinydns loads its data file. An unpatched version of tinydns will open and `mmap` the data file upon every query it receives. If a lot of queries are expected, this behavior decreases performance. The patch alters this behavior by making tinydns only reload the data file at maximum once every second.

[5]This patch adds the line `response_nxdomain();` on line 60 of `server.c`.

[6]This patch is written by Lennert Buytenhek and can be found at `http://tinydns.org/one-second.patch`.

2. **Forwarding Name Server**
   This machine will run the DNSCurve capable forwarding name server, that was developed during this master project. In the benchmark several versions of the name server will be used, i.e. sometimes an implementation that has a different or no cache algorithm. In the actual benchmarks it will be explicitly mentioned what version is used at that time.

3. **Benchmark**
   The benchmark machine houses all the used benchmarking tools. All benchmarks are run on this machine. This means that dnsprepare, dnsbenchmark, and dnsperf are located on this machine.

Properties

The hard- and software that is used, is clear by now. It is therefore time to discuss the used properties during the benchmarks. Figure 5.5 illustrates the topology and line traces of the benchmark setup. These traces are heavily used in the upcoming texts.

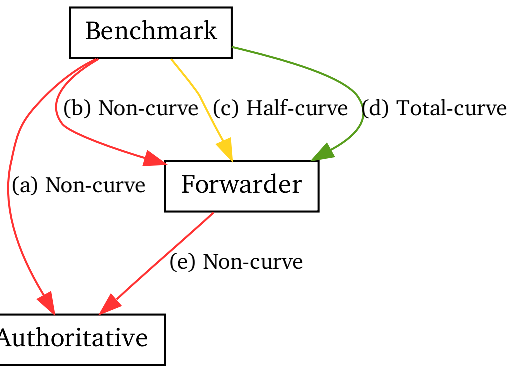

Figure 5.5: *Topology and line traces of benchmark setup*

The three different systems can be seen. In between, logical connections are made. These edges are not truly connections, but so called traces. They show what traffic will be directed over them. Five different traces have been distinguished.

All queries that are part of trace (a) are regular DNS queries that originate from the benchmark server and are destined for the authoritative name server. Trace (b) consists of only regular DNS queries, from the benchmark machine towards the forward server. Trace (c) on the other hand consists of 50% DNSCurve queries, while 50% of the queries do not use DNSCurve. In trace (d) all queries are DNSCurve queries. Finally, trace (e) is distinguished, that only consists of regular DNS queries – which is obvious, since its the forwarder's task to 'translate' DNSCurve queries to standard DNS equivalents. Note that trace (e) is only there for generality, no measures will be done on this trace.

To be able to query the authoritative name server, zone data is needed. It was already told that NIST supplies anonymized zone data [5] that can be used for purposes like these. In the performance tests a combination of NIST's largest TLD zone, together with the largest second level zone is used. Figure 5.6 shows statistics about the tinydns zone file that will be used in all benchmarks.

```
Size on disk (bytes):    5,095,802
Total number of records:   128,387

NS-type records:           106,487
A-type records:             15,744
TXT-type records:            3,047
CNAME-type records:          2,055
MX-type records:             1,052
SOA-type records:                2
```

Figure 5.6: *Statistics of the zone data used in all benchmarks*

The model file has been introduced in the benchmark tools subsection. A model file is the source of an actual traffic load and defines the likelihood of specific queries. The model file that is used as a template by all benchmarks is shown in Figure 5.7.

```
# settings: #MinsToRun #qps
settings: 5 5000

# dnscurve: #DiffKeysToUse PublicKeyServerInBase32
dnscurve: 500 uz5dkhv40x2ym...

# type %ofTotal %nameError %DNSCurve
1 0.70 0.15 0
2 0.06 0.05 0
5 0.08 0.10 0
6 0.02 0.1 0
15 0.12 0.05 0
16 0.02 0.1 0
```

Figure 5.7: *Template model file used by all benchmarks*

Note this is only a template. Nevertheless, resource record type percentages will never change during the benchmarks. The likelihood of name errors, DNSCurve queries, or the number of different keys will however change. All differences between this template model and the actual used model, will always be explicitly mentioned when the results are discussed.

Note when a benchmark is running, no other tasks are being performed. This means that all resources and capacity of the systems are available for the benchmark.

5.5 Results

This section will describe the actual results of the performance tests. Remark that in every of these subsections the traces described by Figure 5.5 will be used to indicate what line is exactly tested.

Seen from the outside, a forwarding name server will in a sense 'replace' a current authoritative name server. However, to fully replace an authoritative name server a data manager must be sure that it is still able to fulfill the reliability and stability criteria. Therefore, four different benchmarks have been devised, that will be discussed in the upcoming subsections.

5.5.1 Regular DNS Performance

Besides translating DNSCurve queries to regular DNS equivalents, the forwarding name server will also have to answer standard DNS queries. Because of minor DNSCurve deployment at this moment, this will be the major task of the forwarding name server in the beginning. Therefore, a good comparison is needed between the performance of the authoritative name server and the forwarding name server.

Specification

The dnsperf [48] tool will be used to execute this benchmark. The input traffic file will be modeled according to the template model of Figure 5.7. However, this time all queries are supposed to be valid, i.e. all name error percentages are 0. Each individual dnsperf benchmark will run for one minute. The data file used by dnsperf will be used multiple times if more queries are sent than there have been made available by the file.

This test is executed over trace (a) and trace (b), where trace (a) is considered to be the maximum achievable result. The following command shows the call to the dnsperf binary:

```
./dnsperf -d ./dnsperf.txt -s IP -b 32 -q 75 -t 2 -l 60
```

The traffic load comes from the `dnsperf.txt` file and target of the attack is the IP listed after `-s`. A send and receive buffer of 32KB will be used and there can be 75 simultaneous outstanding queries. A connection is considered 'lost' when no answer is given within 2 seconds. And – as told – the benchmark will run for 60 seconds.

Each trace will be tested 12 times. This will result in twelve qps measurements for each trace, including a completeness percentage. This percentage indicates when a query timeouts it is considered lost, meaning that all completed queries are the number of received responses.

From all twelve measurements, the largest and smallest value are removed and from these remaining 10 values the average is taken.

Results

The results are represented in Table 5.3. Figure 5.8 illustrates the query rate performance graphically.

	Trace (a)	Trace (b)
Average queries per second:	12,416	10,892
Completeness percentage:	99.91%	99.92%

Table 5.3: *Regular DNS performance and completeness between trace (a) and trace (b)*

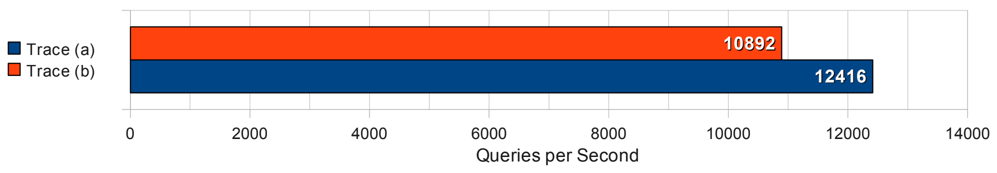

Figure 5.8: *Regular DNS performance between trace (a) and trace (b)*

Discussion

As expected, the trace that passes through the forwarding name server is slower than the one directly sent to the authoritative name server. Meaning that the forwarding name server can handle 87.72% of the non-DNSCurve queries that the authoritative name server can handle directly.

Trace (a) did loose some more packets than trace (b) did – although a negligible amount of 0.01%. This is probably because trace (b) is slower, this makes less opportunity for errors.

5.5.2 Comparative DNS and DNSCurve Performance

Now that the regular DNS performance of the forwarding name server is covered by a specific benchmark, it is time to focus on DNSCurve performance. Because the dnsperf tool does not support DNSCurve, it was decided to develop an own implementation. The two Python programs dnsprepare and dnsbenchmark have been written for this purpose.

The dnsprepare tool was implicitly already used in the previous benchmark, to generate the traffic load for dnsperf. In this benchmark it will be used again to generate DNS and DNSCurve traffic loads.

Calibration Specification

The performance of DNSCurve will be measured relatively against the standard DNS performance. Starting point of this benchmark is trace (a). This trace gives the highest query rate, since it has no entities (such as the forwarding name server) in between. However, over this trace no DNSCurve measurement can be made. Therefore, it will be used as a calibration unit for the dnsbenchmark tool.

This calibration is needed, since dnsbenchmark is relatively inaccurate – especially when the query rate is set high, i.e. above 4,000 qps – because it has to send a specific number of queries per second and simultaneously measure this. Timing is important in this process, but dependent on many factors like the kernel's scheduler, the number of time interrupts a system can generate, and the busyness of both the system and network. Therefore, first a calibration is made that will relate the artificial specified number of queries per second and the actual achieved result.

Trace (a) will act as source for this calibration, as this is the maximum achievable result inside this benchmark setup. The achieved number of qps for this trace will be seen as the maximum result for further measurements in this benchmark. Notice that this calibration should be conducted for each and every artificial specified qps.

An example of this calibration. Let's say dnsbenchmark is run on trace (a) with the qps set at 1,000. After the benchmark it appears the actual achieved qps was 900. Later measurements at 1,000 qps will now be related to this 900 qps. So if the 1,000 qps benchmark is done for trace (c) and the result is 750 qps, a performance of 83% is achieved instead of 75%.

For the calibration – as well as the real benchmarks afterwards – the query rate is explicitly set at 250 qps for the first run. Every next run, 250 qps will be added. In each run, three different measurements are made that all last for 30 seconds. These measurements are however not executed after each other, but at different times. The average value of the three measurements is taken as the actual reached result for this specific run. Before each individual measurement, the tinydns daemon at the authoritative name server will be stopped and started.

Calibration Result

Figure 5.9 shows the chart of this calibration step. The blue bars indicate the number of actual reached query rates, the numbers inside the bar show the actual achieved query rate. While the remaining red bars show what the theoretical specified query rate should have been. The values on the X-axis indicate the artificial query rate that was specified and the yellow line (that uses the right Y-axis) indicates the achieved accuracy. Remark that these results can also be found in table form in Table A.1.

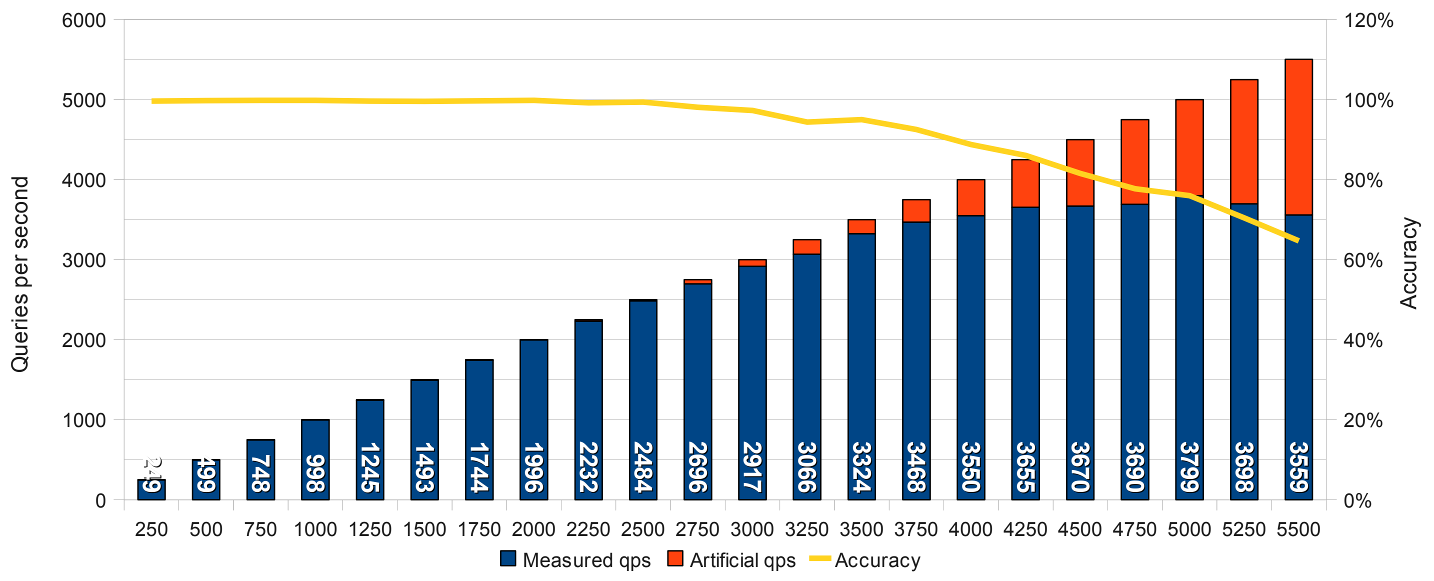

Figure 5.9: *Calibration chart of trace (a) to relate measured qps to artificial qps*

Calibration Discussion

Notice the blue bars and the yellow line tell totally different stories. The blue bars indicate the maximum reached query rate (i.e. performance) of the authoritative name server. It can be seen the maximum rate (3,799) is reached at an artificial rate of 5,000 queries. After this, the query rate performance stabilizes (or: saturates).

The yellow line on the other hand shows the accuracy of the dnsbenchmark tool, i.e. the need for calibration. The hypothesis that accuracy decreases when the query rate increases, was right.

Note the maximum reached query rates from the dnsbenchmark tool are remarkably lower than the ones achieved by dnsperf in the previous subsection. The main reason for this is the language in which the tools are written. dnsperf is written in C, while dnsbenchmark is written in Python. Although dnsbenchmark uses threads in order to process as many queries and responses simultaneously, Python has the limitation it can only use one CPU at a time[7], making the threads relatively useless. Notice however, this performance drop does not influence this benchmark, because all results obtained by the dnsbenchmark tool will have this performance drop included.

[7]This is due to Python's design. In particular, it is caused by the so called Global Interpreter Lock. It goes however beyond the scope of this thesis to go into detail.

Specification

Now that there are calibrated maximum achievable query rates, actual benchmarking of the forwarding name server traces can start. Just like in the calibration step, for each run three different measurements will be made at different time intervals (i.e. not directly after each other). Each measurement will run for 30 seconds. A special version of the forwarding name server is used that does *not* include a shared secret caching mechanism. This means that for each DNSCurve query it receives, it has to compute a shared secret to decrypt the packet and to encrypt the eventual response. The caching algorithm itself will be tested in the next subsection's benchmark. Before each individual measurement, both tinydns and the forwarding daemon will be stopped and started.

The traffic load for trace (b) consists of only regular DNS queries, while trace (c) and trace (d) include 50 and 100% DNSCurve queries respectively. The measurement starts at 250 qps and will increase every step with 250 qps, this stops at 5,500 qps. Meaning that in total 22 runs will be made (just like in the calibration).

Results and Discussion

First, the absolute measurements will be shown. Notice that this measurement has nothing to do with the previous calibration step, it shows the achieved query rates for the different traces. Also the results of trace (a) from the calibration step are included. Figure 5.10 illustrates the absolute measurements. These results can also be found in table form in Table A.2.

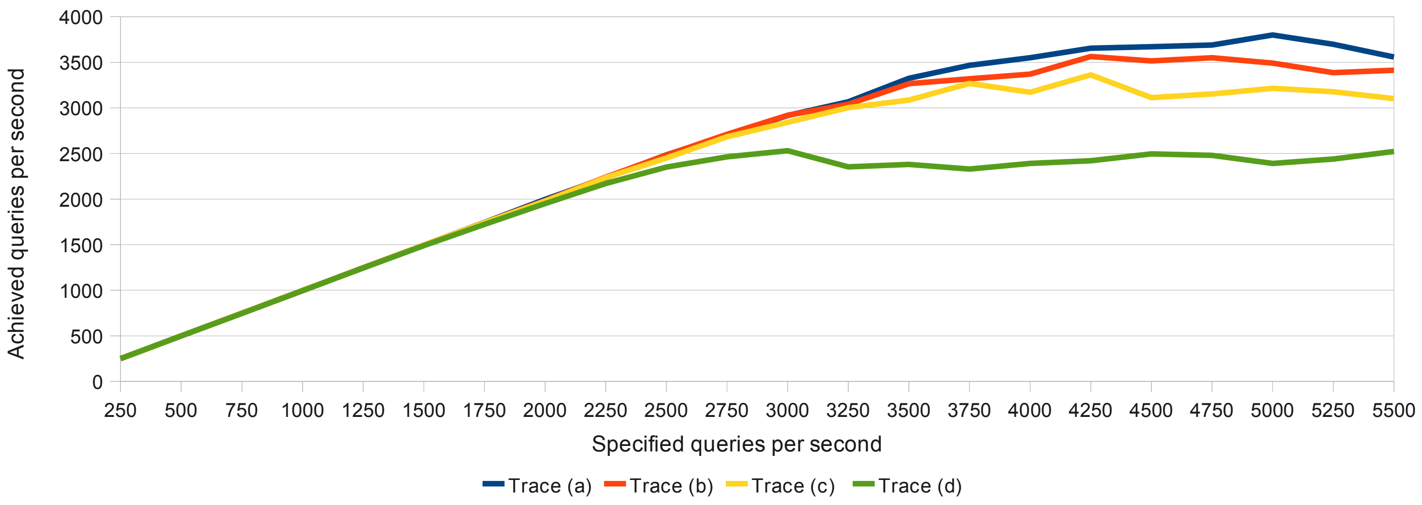

Figure 5.10: *Absolute performance of traces (a), (b), (c), and (d)*

Notice the blue line shows trace (a), this is also the calibration measurement, i.e. it is considered the maximum achievable query rate for non-DNSCurve queries in this benchmark setup. The non-DNSCurve trace (b) that goes through the forwarder is closely related to the one of trace (a). The DNSCurve computations the forwarder needs to perform for traces (c) and (d) can be noticed by a degrading performance.

Let's relate these measurements to the calibration trace. Figure 5.11 portrays the achieved relative performance for trace (b) to trace (d). Note the percentage is related to the calibrated query rate of non-DNSCurve queries over trace (a) (i.e. the values inside the blue bars of Figure 5.9). Also these results can be found in table form in Table A.2.

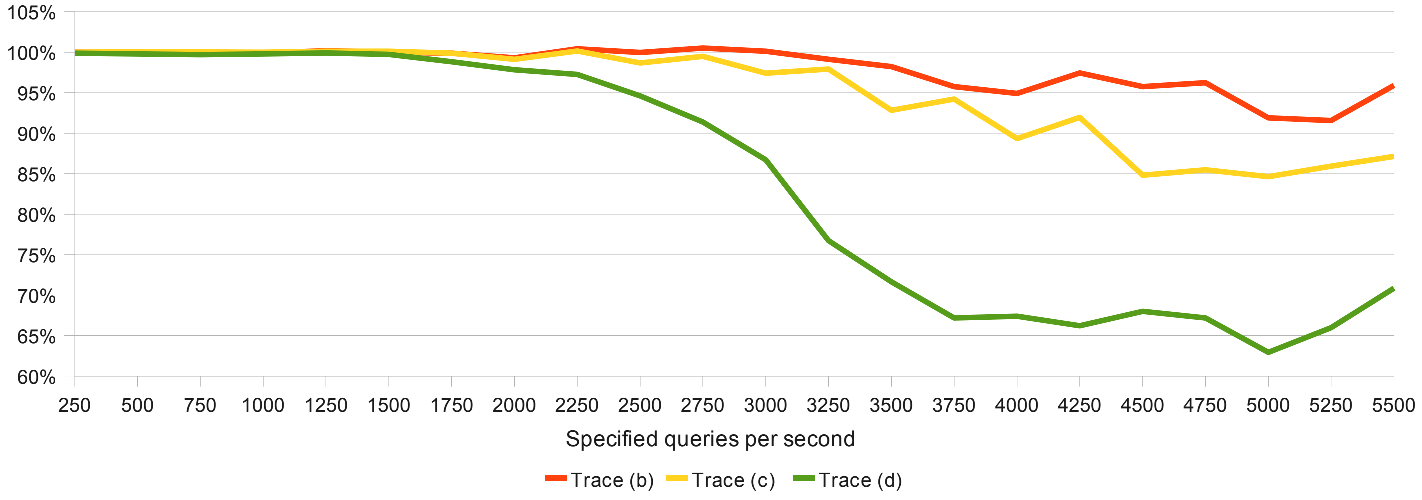

Figure 5.11: *Performance of all forwarder related traces relative to trace (a)'s performance*

The red line (trace (b)) shows the performance of the forwarding name server, when only non-DNSCurve are sent. It can be seen that when the query rate is increased, performance of the forwarding name server slowly decreases.

The yellow line (trace (c)) indicates the performance when 50% of the queries are DNSCurve queries. Up to 3,000 qps the performance is comparable with trace (b)'s performance, after this point, the performance drops further until around 85%.

Finally, the green line (trace (d)) represents the performance when all queries are DNSCurve queries. Until 1,750 qps the performance is comparable with both previous traces, however afterwards the performance drops significantly. At faster query rates, the performance stabilizes around 65%.

The next benchmark will benchmark shared secret caching algorithms, that should improve both trace (c) and trace (d)'s performance drastically. Since not for each client public key that already came by, a new shared secret has to be computed.

5.5.3 Shared Secret Caching Performance

In the previous benchmark the DNSCurve performance was tested for the first time. It could be seen that it performed not well in comparison with non-DNSCurve queries. Hypothesis is that a shared secret caching algorithm will positively influence the forwarder's performance when dealing with DNSCurve queries.

This benchmark will focus on confirming or denying this hypothesis. From the technology section it is known only one caching algorithm is implemented in the forwarder so far. Nevertheless, this implementation will be tested by this benchmark.

Specification

This benchmark looks a lot like the previous one. In fact, the calibration made by that benchmark will be used again as a calibration unit in this benchmark. So the query rate achieved by queries over trace (a) (i.e. non-DNSCurve, directly to the authoritative name server) is considered to be the maximum achievable query rate in this benchmark too.

This benchmark will only perform one new test. This result will be related with the achieved query rate over trace (b) and trace (d) from the previous benchmark. Trace (b) (no DNSCurve queries, through forwarder) since it is considered to be the maximum achievable that passes through the forwarding name server. And trace (d) because it consists solely of DNSCurve queries, and showed the performance of the forwarding name server without any shared secret caching algorithm. So the only new benchmark is done over trace (d), although this time with a hash table cache algorithm in place.

The test methodology is a bit different. Instead of letting each benchmark run for 30 seconds, in total 50,000 queries will be send. This is done to prevent that benchmarks with faster query rates will be able to cache more DNSCurve keys, because they would reach a key earlier in the traffic load. This directly implies that the query rate will still be artificially fixed. Note that the two old traces (i.e. trace (b) and (d)) can still be used. Since trace (b) did not involve any DNSCurve queries and because the forwarding name server did not cache any shared secrets in the previous benchmark, trace (d) could not gain any benefit of it. Before each individual measurement, both tinydns and the forwarding daemon will be stopped and started.

There is room for 1,000 shared secrets, the hash table itself consists of 40 buckets. Since there are only 500 different DNSCurve public keys (see the template model of Figure 5.7) this means there is enough room to store each shared secret.

Results and Discussion

First, the absolute measurements will be shown. It shows the achieved query rates for the different traces, also the results of trace (a) – known from the calibration step – are included. Figure 5.12 illustrates the absolute measurements. Note that the actual performance results can also be found in table form in Table A.3.

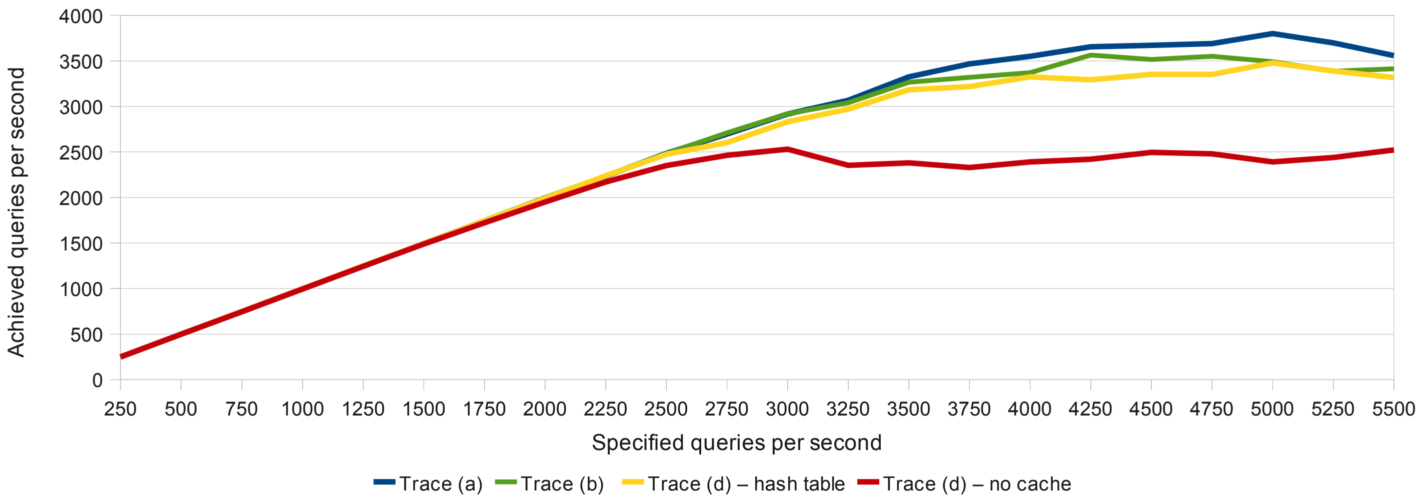

Figure 5.12: *Absolute query rates of the traces (a), (b), (d) without a cache, and (d) with a cache in place*

Notice the blue line shows trace (a), this is also the calibration measurement, i.e. it is considered the maximum achievable query rate for non-DNSCurve queries in this benchmark setup. The non-DNSCurve trace (b) that goes through the forwarder is closely related to the one of trace (a). The red line, indicating the performance of trace (d) without the cache algorithm in place shows a quickly degrading performance, that could also be seen in Figure 5.10. The yellow line shows trace (d) with the cache algorithm in use. It can be seen this trace now performs as expected, in between the trace (b)'s performance and the no cache algorithm performance.

From the previous benchmark it is known that when the query rates get higher, accuracy of the benchmark decreases. It can be seen that trace (d)'s cache performance almost outperforms the non-DNSCurve when the query rate is around 5,000 qps. This is considered to be caused by dnsbenchmark's inaccuracy at higher query rates.

Figure 5.13 portrays the achieved relative performance for trace (b), and both traces of (d). Note the percentage is related to the calibrated query rate of non-DNSCurve queries over trace (a). The actual performance results can also be found in table form in Table A.3.
The results of traces (b) and (d) without the cache in place have already been discussed in the previous benchmark. It can be seen that the perfor-

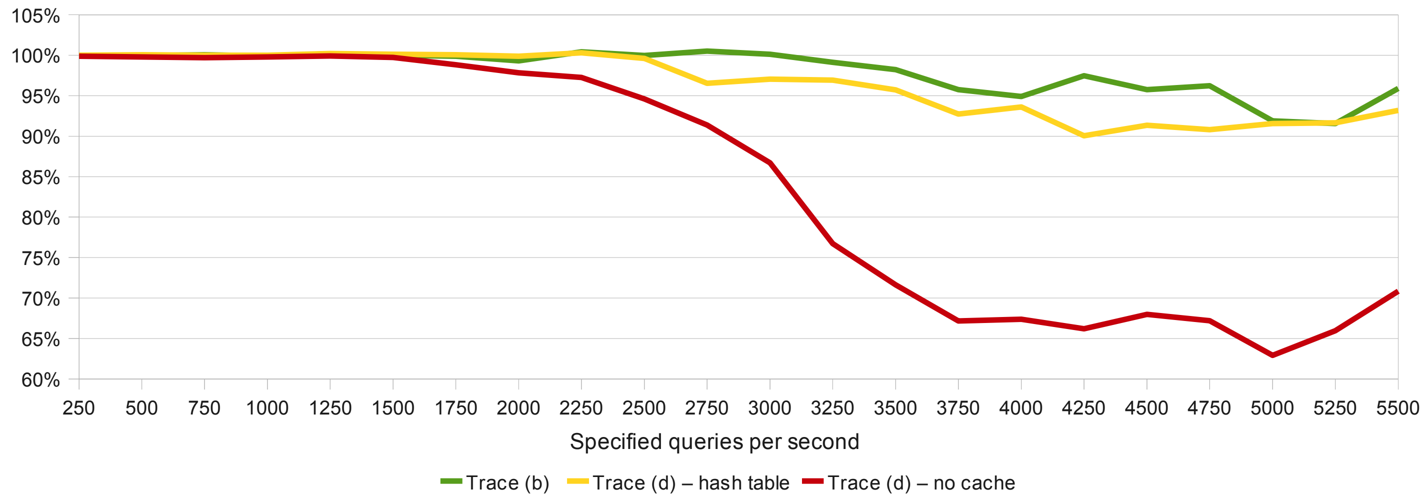

Figure 5.13: *Relative performance of the traces (b), (d) without a cache, and (d) with a cache in place*

mance of trace (d) with a cache in place has been drastically improved. It can almost be related towards the performance the performance of the non-DNSCurve trace (b).

Note that the specified number of shared secrets that could be cached in this benchmark was very low (although enough to cache all the public keys and their associated shared secrets for the benchmark). Only 88,256 bytes of memory had to be allocated to be able to store 1,000 DNSCurve keys. (Notice this number is dependent on what CPU architecture (i.e. 32- or 64-bit) is used.) The cache could therefore easily be scaled to store many more shared secrets.

5.5.4 CPU Usage Benchmark

The benefits of a cache are known by now. However, before a shared secret can be put into this cache, it first has to be computed. From the DNSCurve chapter it is already known this is the most intensive operation in a DNSCurve conversation. The CPU exhaustion attack tries to exploit this fact.

This benchmark will show how a DNSCurve capable system behaves when it constantly receives DNSCurve queries and is not able to make use of a shared secret cache.

Specification

There are several ways to determine, but also measure, CPU usage in Linux. For this benchmark, the `sar(1)` tool will be used. `sar(1)` is part of the `sysstat` package, that contains various system statistics tools. `sar(1)` “collects, reports and saves system activity information (CPU, memory, disks, interrupts, network interfaces, TTY, kernel tables, etc.)” [36]. In this benchmark only CPU usage will be considered.

Linux distinguishes five categories in which the CPU can be active. Measurements for these five categories are given relatively, i.e. for each category it will report how many percent of the time the CPU has been utilized for tasks in this category. The `user` category represents the CPU utilization that occurred while executing tasks at user level (i.e. the application), the `nice` category indicates CPU utilization while executing tasks at user level that have the nice property set – this is a way to set priority for Unix processes. The `system` category indicates CPU utilization while executing tasks at the system level (i.e. executing kernel tasks), the `iowait` category indicates the CPU was idle because it had to wait for input to arrive or output to be send. The last category shows all the percentages that are left, i.e. the time the CPU was idle and did not had to wait for I/O. Note this story applies to systems with more CPUs – like in this benchmark setup, where every guest host has the ability to use two (virtual) CPUs – too. Also remark that categories are mutually exclusive, i.e. their usage together will always be 100%.

This benchmark will focus on the `user`, `system`, and `iowait` category. Although the forwarding name server implementation has no influence on usage of the `system` category, it will show what other usage is caused by the kernel.

The benchmark itself differs from the previous benchmarks. Trace (b), trace (c), and trace (d) will however again be used. Only this time, there is a time limit of 5 minutes per run. This means the dnsbenchmark tool will run for 5 minutes straight. So if the end of the traffic load file is reached, it will be run again.

For each trace, this benchmark is executed three times. Meaning that in total 15 minutes of data will be gathered per trace, `sar(1)` will give an average utilization value for each of the categories. The average value for all three traces will be the result, categorized by the three different categories, together with a summation of all three values.

The artificial query rate will still be set. It is set on the rate that gave the highest query rate in all traces: 4,250 qps. There is *no* caching algorithm used by the forwarding name server. This means that every shared secret has to be computed, for each of the DNSCurve queries it receives. Note that this behavior can be compared with a dishonest client, that is sending rogue DNSCurve query packets, containing random public keys and/or garbage inside the packet.

Results

The results are represented in Table 5.4. Figure 5.14 illustrates the CPU utilization graphically.

	user %	system %	iowait %	total %
Trace (b)	0.87%	6.44%	8.87%	**16.19%**
Trace (c)	24.44%	14.57%	2.46%	**41.47%**
Trace (d)	34.98%	14.76%	0.07%	**49.81%**

Table 5.4: *Average categorized CPU utilization during execution of traces (b), (c), and (d)*

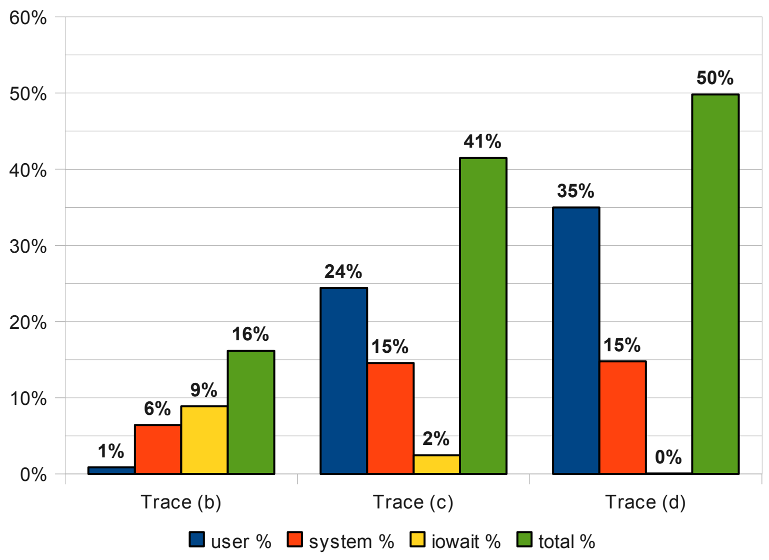

Figure 5.14: *Average categorized CPU utilization during execution of traces (b), (c), and (d)*

Discussion

It can be seen that forwarding non-DNSCurve queries (trace (b)) is not a big deal for the forwarding name server. Only 16% percent of the CPU is used, when no DNSCurve related work has to be done. Note that the application itself (the `user` category) needs less than 1% of the CPU. Most CPU time is spend on waiting for I/O, meaning that forwarding regular DNS queries is I/O bound.

When half of the queries are DNSCurve queries (i.e. trace (c)), the story changes drastically. In total more than 41% of the CPU will be used. The kernel is double as busy (the `system` category) compared with the previous trace, but most interesting is the increase of application CPU utilization. More than half of the total utilization is claimed by the forwarding name server itself. Note also that less time is spend on waiting for I/O, this means that when DNSCurve comes into play, the forwarding name servers gets CPU bound.

When trace (d) is considered (that consists entirely of DNSCurve queries) it can be seen that almost half of the CPU is used and more time is claimed by the forwarding name server. Which is declarable, since it has to compute a lot more shared secrets per second. The kernel is about as busy, as it was with trace (c), and it can be seen that even less CPU time is spend (a negligible 0.07%) on waiting for I/O. This indicates the more DNSCurve related work there is to be done, the more the forwarding name server gets CPU bound. Nevertheless, when the outcome of the previous benchmark is considered, even bigger optimizations can be obtained when much memory is available to store shared secrets computed along the way.

Chapter 6

Conclusion

The Domain Name System suffers several security related vulnerabilities that make an attacker able to control parts of the information flow the DNS facilitates. The most significant attack, the so called active cache poisoning attacker, can provide the attacker a way to insert false DNS responses. The client will not be able to distinguish these false responses from real ones. With this attack the attacker would be able to, for example hijack someone's email and determine ones login credentials. Therefore, DNSSEC has been introduced.

DNSSEC uses public key cryptography to solve many of the problems the DNS suffers, i.e. it provides data integrity and authenticity for the DNS. Note however that DNSSEC is not able to stop the attacks, it makes them detectable by using digital signatures. Advantage of DNSSEC is the fact that it is officially standardized and implementation feasibility studies have been performed, that second DNSSEC is practically implementable. Global deployment is currently taking place, for example the root name servers and some major TLDs are DNSSEC capable by now.

Besides extra added protection, DNSSEC also introduces new problems that are security related. The most severe problems include the possibility to conduct replay-, amplification-, and zone enumeration attacks. Due to DNSSEC offline generation of signatures, freshness of DNSSEC packets can not be guaranteed. Therefore, DNSSEC packets can be replayed for a limited time period (usually one month), resulting in integrity and availability issues. Besides, DNSSEC greatly increases the likelihood of amplification attacks, because it introduces new resource record types that are significantly larger than existing RR types. With these amplification attacks, an attacker is able to attack any host's or network's availability, even if such hosts are not DNS related. DNSSEC also makes zone enumeration much easier when the NSEC-type denial of existence record is used. With such attack an attacker can determine all response names of a zone.

DNSCurve was developed independently of DNSSEC, its design goal however, is the same: solving the problems the regular DNS suffers. DNSCurve also uses public key cryptography to provide confidentiality, integrity, authenticity, and limited non-repudiation. Instead of only using signatures, DNSCurve uses a combination of new high-speed cryptographic primitives to detect all known attacks. It is not officially standardized yet, although there is a draft standard submitted. No public implementations exist yet, implying that deployment has not started. Due to the fact that DNSCurve is not (and probably never will be) adopted by the root servers and TLD name servers, the first iterations of a traversal will never be secured. To be able to achieve an entirely secure traversal, a hybrid combination with DNSSEC is needed. That is possible due to the different approach of both protocols.

DNSSEC focuses on protecting the object (i.e. DNS zone data), while DNSCurve protects the communication channel. DNSSEC does this by changing the regular DNS' semantics, providing end-to-end security. DNSCurve on the other hand, does not interfere with the standard DNS, it only provides link level security. Disadvantage of this approach is that any intermediate DNSCurve capable host used in a traversal can alter DNS packets without the client knowing, this is known as hop-to-hop security. An advantage is that DNSCurve does not care what kind of DNS packets it protects, as long as it are valid DNS packets. This means that DNSSEC can be used inside DNSCurve packets, making this hybrid approach possible. This indirectly gives a solution to the situation in which the initial iterations of a DNSCurve traversal are not secured: by using DNSSEC for the top levels of a traversal.

DNSCurve does not introduce any of the problems DNSSEC introduced. No replay attacks, because it uses realtime encryption and decryption; no significant increase in bandwidth usage to become a viable amplification attack candidate; and it leaks no more privacy related information than the regular DNS already does.

During the development process of a DNSCurve capable forwarding name server, it was tested whether this implementation could keep up with a regular DNS implementation. Benchmarks show the forwarding name server is able to process around 87% of the standard DNS queries a standalone authoritative name server would be able to handle. When half of all packets would be DNSCurve queries, the forwarding name server's performance would drop with 15%, if no shared secret caching algorithm was used. This percentage is relative to the non-DNSCurve performance when regular DNS queries were directly sent to the authoritative name server. If all packets would be DNSCurve queries, performance drops with nearly 25%. However, the usage of a shared secret caching algorithms greatly improves the performance. So well that it can almost be compared with non-DNSCurve performance. DNSCurve computations are CPU bound, nevertheless when more shared secrets can be cached because much memory is available, this gives even better optimizations than buying a new faster CPU.

All in all, this makes DNSCurve a feasible extension for the regular DNS, seen from a performance point of view. The lack of root-, and TLD name server support will still require DNSSEC to guarantee a fully secure traversal. Disadvantage of this hybrid approach would be the double cryptographic work for a client. The only benefit for a client in this approach would be query name confidentiality.

Appendix A

Performance Results

A.1 Comparative DNS and DNSCurve

	Calibration trace (a)	
Artificial qps	**Avg measured qps**	**Accuracy**
250	249	99,60%
500	499	99,73%
750	748	99,78%
1000	998	99,77%
1250	1245	99,57%
1500	1493	99,56%
1750	1744	99,68%
2000	1996	99,78%
2250	2232	99,19%
2500	2484	99,36%
2750	2696	98,05%
3000	2917	97,23%
3250	3066	94,35%
3500	3324	94,98%
3750	3468	92,47%
4000	3550	88,75%
4250	3655	86,01%
4500	3670	81,56%
4750	3690	77,68%
5000	3799	75,99%
5250	3698	70,44%
5500	3559	64,70%

Table A.1: *Calibration measurement on trace (a)*

	Trace (b)		**Trace (c)**		**Trace (d)**	
Artificial qps	**Avg measured qps**	**Diff**	**Avg measured qps**	**Diff**	**Avg measured qps**	**Diff**
250	249	100.00%	249	100.00%	249	99.87%
500	498	99.93%	499	100.07%	498	99.80%
750	749	100.04%	749	100.04%	746	99.69%
1000	997	99.90%	998	100.00%	996	99.80%
1250	1247	100.19%	1246	100.11%	1244	99.92%
1500	1493	100.00%	1495	100.13%	1489	99.73%
1750	1742	99.87%	1742	99.89%	1724	98.83%
2000	1982	99.30%	1978	99.13%	1952	97.83%
2250	2241	100.42%	2235	100.15%	2171	97.27%
2500	2483	99.97%	2451	98.67%	2350	94.61%
2750	2710	100.52%	2683	99.49%	2464	91.40%
3000	2921	100.13%	2841	97.41%	2529	86.70%
3250	3040	99.13%	3003	97.92%	2353	76.74%
3500	3265	98.22%	3086	92.84%	2381	71.63%
3750	3320	95.74%	3267	94.21%	2329	67.17%
4000	3369	94.91%	3171	89.32%	2392	67.38%
4250	3562	97.46%	3362	91.97%	2420	66.20%
4500	3514	95.76%	3113	84.81%	2495	67.98%
4750	3551	96.23%	3154	85.48%	2479	67.19%
5000	3491	91.88%	3215	84.63%	2390	62.91%
5250	3386	91.57%	3178	85.93%	2440	65.98%
5500	3412	95.89%	3101	87.13%	2522	70.86%

Table A.2: *Performance of traces (b), (c), and (d) in relation with the calibrated values*

A.2 Shared Secret Caching

		Trace (d) no cache		Trace (d) hash table	
Artificial qps	**Calibrated max qps**	**Avg measured qps**	**Diff**	**Avg measured qps**	**Diff**
250	249	248.7	99.87%	249.0	100.00%
500	499	497.7	99.80%	499.0	100.07%
750	748	746.0	99.69%	748.3	100.00%
1000	998	995.7	99.80%	997.7	100.00%
1250	1245	1243.7	99.92%	1247.3	100.21%
1500	1493	1489.3	99.73%	1495.0	100.11%
1750	1744	1724.0	98.83%	1745.3	100.06%
2000	1996	1952.3	97.83%	1993.0	99.87%
2250	2232	2170.7	97.27%	2238.3	100.30%
2500	2484	2350.0	94.61%	2474.3	99.61%
2750	2696	2464.3	91.40%	2603.0	96.54%
3000	2917	2529.0	86.70%	2830.3	97.03%
3250	3066	2353.0	76.74%	2971.7	96.91%
3500	3324	2381.3	71.63%	3182.0	95.72%
3750	3468	2329.3	67.17%	3216.0	92.74%
4000	3550	2392.0	67.38%	3323.3	93.62%
4250	3655	2420.0	66.20%	3291.7	90.05%
4500	3670	2495.0	67.98%	3352.7	91.35%
4750	3690	2479.0	67.19%	3351.0	90.82%
5000	3799	2390.3	62.91%	3478.7	91.56%
5250	3698	2440.0	65.98%	3388.7	91.64%
5500	3559	2521.7	70.86%	3316.3	93.19%

Table A.3: *Shared secret caching algorithm performance in relation with calibrated values*

Bibliography

[1] DNS Extensions (dnsext) Working Group. http://datatracker.ietf.org/wg/dnsext/charter, date last accessed June 9, 2010. 38

[2] IANA – Root Zone Database. http://www.iana.org/domains/root/db, date last accessed 2 March 2010. 13

[3] DNS Statement of Policy, June 1998. http://www.ntia.doc.gov/ntiahome/domainname/6_5_98dns.htm, date last accessed 21 May 2010. 2, 8, 9

[4] DNS Performance Test (DPT), Sept. 2007. http://swmirror.org/drupal/node/92, last accessed July 22, 2010. 132

[5] NIST – Domain Name System Security (DNSSEC) Project, Apr. 2008. http://www-x.antd.nist.gov/dnssec, date last accessed July 2, 2010. 2, 60, 109, 133, 140

[6] DNSCurve: Usable security for DNS, July 2009. http://www.dnscurve.org, date last accessed June 29, 2010. 2, 68, 71, 72, 76, 77, 78, 79, 86, 87, 89, 97, 98, 105, 110, 119

[7] NaCl: Networking and Cryptography library, Mar. 2009. http://nacl.cace-project.eu, date last accessed 24 March 2010. 85, 88, 117, 123, 138

[8] DNSSEC – The DNS Security Extensions – Protocol Home Page, May 2010. http://www.dnssec.net, date last accessed June 9, 2010. 2, 35, 39

[9] List of DNS record types. Wikipedia, June 2010. http://en.wikipedia.org/wiki/List_of_DNS_record_types, last accessed July 12, 2010. 2, 134

[10] OpenDNSSEC, 2010. http://www.opendnssec.org, date last accessed June 25, 2010. 61, 110

[11] ABLEY, J., AND SCHLYTER, J. Root Zone Trust Anchor Publication, May 2010. `http://data.iana.org/root-anchors/draft-icann-dnssec-trust-anchor.txt`. 51

[12] ARENDS, R., AUSTEIN, R., LARSON, M., MASSEY, D., AND ROSE, S. RFC 4033: DNS Security Introduction and Requirements, Mar. 2005. 2, 35, 39, 41, 72, 110

[13] ARENDS, R., AUSTEIN, R., LARSON, M., MASSEY, D., AND ROSE, S. RFC 4034: Resource Records for the DNS Security Extensions, Mar. 2005. 2, 37, 39, 42, 47, 110

[14] ARENDS, R., AUSTEIN, R., LARSON, M., MASSEY, D., AND ROSE, S. RFC 4035: Protocol Modifications for the DNS Security Extensions, Mar. 2005. 2, 39, 56, 57, 58, 69, 110

[15] AUSTEIN, R., AND ATKINS, D. RFC 3833: Threat Analysis of the Domain Name System (DNS), Aug. 2004. 2, 24

[16] BELLOVIN, S. M. Using the Domain Name System for System Break-Ins. In *Proceedings of the Fifth Usenix Unix Security Symposium* (Salt Lake City, UT, June 1995), pp. 199–208. 2, 38

[17] BERNSTEIN, D. J. djbdns: Domain Name System tools. `http://cr.yp.to/djbdns.html`. 76, 135, 138

[18] BERNSTEIN, D. J. The qmail security guarantee, Mar. 1997. `http://cr.yp.to/qmail/guarantee.html`. 76

[19] BERNSTEIN, D. J. The Poly1305-AES message-authentication code, Mar. 2005. `http://cr.yp.to/mac/poly1305-20050329.pdf`. 87

[20] BERNSTEIN, D. J. Curve25519: new Diffie–Hellman speed records. In *Proceedings of PKC 2006* (2006), M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., Lecture Notes in Computer Science 3958, Springer, pp. 207–228. 86

[21] BERNSTEIN, D. J. The Salsa20 family of stream ciphers, Dec. 2007. `http://cr.yp.to/snuffle/salsafamily-20071225.pdf`. 87

[22] BERNSTEIN, D. J. DNSCurve: Usable security for DNS. Slides of talk, Aug. 2008. `http://cr.yp.to/talks/2008.08.22/slides.pdf`. 70

[23] BERNSTEIN, D. J. Cryptography in NaCl, Mar. 2009. `http://cr.yp.to/highspeed/naclcrypto-20090310.pdf`. 86, 88

[24] BERNSTEIN, D. J. djbdns$\leq$1.05 lets AXFRed subdomains overwrite domains, Mar. 2009. `http://marc.info/?l=djbdns&m=123613000920446&w=2`. 76

[25] CHANDRAMOULI, R., AND ROSE, S. NIST SP800-81r1: Secure Domain Name System (DNS) Deployment Guide. Tech. rep., NIST Special Publication, Feb. 2009. 50

[26] CONRAD, D. RFC 3225: Indicating Resolver Support of DNSSEC, Dec. 2001. 51

[27] DAVIS, J. Secret Geek A-Team Hacks Back, Defends Worldwide Web. Wired Magazine, Nov. 2008. `http://www.wired.com/techbiz/people/magazine/16-12/ff_kaminsky`. 30, 65

[28] DAVIS, K., TURNER, J., AND YOCOM, N. *The Definitive Guide to Linux Network Programming*, 1st ed. APress, Aug. 2004. 126

[29] DEERING, S., AND HINDEN, R. RFC 2460: Internet Protocol, Version 6 (IPv6) Specification, Dec. 1998. 6, 120, 130

[30] DEMPSKY, M. DNSCurve: Link-Level Security for the Domain Name System, Feb. 2010. `http://tools.ietf.org/html/draft-dempsky-dnscurve-01`. 2, 78, 90, 110, 118

[31] DEMPSKY, M. Matthew Dempsky's (mdempsky) DNSCurve related code at GitHub, June 2010. `http://github.com/mdempsky/dnscurve`. 118, 129

[32] DEMPSKY, M. OpenDNS adopts DNSCurve, Feb. 2010. `http://blog.opendns.com/2010/02/23/opendns-dnscurve`. 131

[33] EASTLAKE, D., BRUNNER-WILLIAMS, E., AND MANNING, B. RFC 2929: Domain Name System (DNS) IANA Considerations, Sept. 2000. 19

[34] FERGUSON, P., AND SENIE, D. BCP 38: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing, May 2000. 32

[35] GIEBEN, R. DNSSEC in NL, Jan. 2004. `http://www.nlnetlabs.nl/downloads/publications/dnssec/dnssecnl/secreg-report.pdf`. 2, 60

[36] GODARD, S. sysstat – Performance tools for Linux, June 2010. `http://sebastien.godard.pagesperso-orange.fr`, last accessed July 7, 2010. 150

[37] HOBBES, R. Hobbes' Internet Timeline, Jan. 2010. `http://www.zakon.org/robert/internet/timeline`, date last accessed 8 June 2010. 38

[38] KEGEL, D. The C10K problem, Sept. 2006. `http://www.kegel.com/c10k.html`. 125

[39] KOLKMAN, O. Measuring the resource requirements of DNSSEC, Sept. 2005. `http://www.ripe.net/docs/ripe-352.html`. 60

[40] KOLKMAN, O., AND GIEBEN, R. RFC 4641: DNSSEC Operational Practices, Sept. 2006. 49

[41] LANGLEY, A. Adam Langley (agl)'s DNSCurve related code at GitHub, Sept. 2008. `http://github.com/agl/dnscurve`. 117, 129

[42] LAURIE, B., SISSON, G., ARENDS, R., AND BLACKA, D. RFC 5155: DNS Security (DNSSEC) Hashed Authenticated Denial of Existence, Feb. 2008. 45, 47, 72, 110

[43] LEHMANN, M. libev – A full-featured and high-performance event loop, July 2008. `http://software.schmorp.de/pkg/libev.html`. 125, 138

[44] MOCKAPETRIS, P. RFC 1034: Domain Names – Concepts and Facilities, Nov. 1987. 2, 9, 16, 38

[45] MOCKAPETRIS, P. RFC 1035: Domain Names – Implementation and Specification, Nov. 1987. 2, 9, 11, 16, 17, 38, 81, 89

[46] MOORE, D. DNS Server Survey, May 2004. `http://mydns.bboy.net/survey`, date last accessed 23 March 2010. 122

[47] NLNET LABS. A short history of DNSSEC. `http://nlnetlabs.nl/projects/dnssec/history.html`, date last accessed June 9, 2010. 2, 38

[48] NOMINUM INC. DNSPerf – DNS performance testing, Jan. 2008. `http://www.nominum.com/services/measurement_tools.php`, last accessed July 17, 2010. 133, 141

[49] OBAMA, B. Remarks by the President on Securing our Nation's Cyber Infrastructure, May 2009. `http://www.whitehouse.gov/the_press_office/Remarks-by-the-President-on-Securing-Our-Nations-Cyber-Infrastructure`. 1

[50] OPENDNS. OpenDNS System, July 2010. `http://system.opendns.com`, last accessed July 24, 2010. 131

[51] OSTERWEIL, E., MASSEY, D., AND ZHANG, L. SecSpider the DNSSEC Monitoring Project, July 2010. `http://secspider.cs.ucla.edu`, date last accessed July 9, 2010. 48, 109

[52] OZMENT, A., AND SCHECHTER, S. E. Bootstrapping the Adoption of Internet Security Protocols. In *Fifth Workshop on the Economics of Information Security* (Cambridge, UK, 2006). 130

[53] POSTEL, J. RFC 768: User Datagram Protocol, Aug. 1980. 24, 31

[54] POSTEL, J. RFC 791: Internet Protocol, Sept. 1981. 5

[55] POSTEL, J. RFC 793: Transmission Control Protocol, Sept. 1981. 24

[56] PROVOS, N., AND MATHEWSON, N. libevent – an event notification library, Apr. 2004. `http://www.monkey.org/~provos/libevent`. 125

[57] RSA DATA SECURITY. PKCS #1: RSA Cryptography Standard. `http://www.rsa.com/rsalabs/node.asp?id=2125`, date last accessed June 17, 2010. 46

[58] STEVENS, W. R., FENNER, B., AND RUDOFF, A. M. *UNIX Network Programming – The Sockets Networking API*, 3rd ed., vol. 1. Addison-Wesley Professional, Nov. 2003. 126

[59] STROMBERG, T. R. namebench – Open-source DNS Benchmark Utility, June 2010. `http://code.google.com/p/namebench`, last accessed July 22, 2010. 132

[60] VIXIE, P. RFC 2671: Extension Mechanisms for DNS (EDNS0), Aug. 1999. 51