Electronic Communications of the EASST
Volume 48 (2011)

Proceedings of the
Fifth International Workshop on
Foundations and Techniques for
Open source Software Certification
(OpernCert 2011)

A Formal Specification of the DNSSEC Model

Ezequiel Bazan Eixarch, Gustavo Betarte, and Carlos Luna

20 pages

Guest Editors: Luis Soares Barbosa, Dimitrios Settas
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Formal Specification of the DNSSEC Model

Ezequiel Bazan Eixarch[1], **Gustavo Betarte**[2], **and Carlos Luna**[2]

[1] ezequielbazan@gmail.com
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario, Rosario, Argentina

[2] [gustun,cluna]@fing.edu.uy
Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Montevideo, Uruguay

Abstract: The Domain Name System Security Extensions (DNSSEC) is a suite of specifications that provide origin authentication and integrity assurance services for DNS data. In particular, DNSSEC was designed to protect resolvers from forged DNS data, such as the one generated by DNS cache poisoning. This article presents a minimalistic specification of a DNSSEC model which provides the grounds needed to formally state and verify security properties concerning the chain of trust of the DNSSEC tree. The model, which has been formalized and verified using the Coq proof assistant, specifies an abstract formulation of the behavior of the protocol and the corresponding security-related events, where security goals, such as the prevention of cache poisoning attacks, can be given a formal treatment.

Keywords: DNS, DNSSEC, security properties, formal modelling, Coq

1 Introduction

The Domain Name System (DNS) [Moc87a] [Moc87b] constitutes a distributed database that provides support to a wide variety of network applications such as electronic mail, WWW, and remote login. The database is indexed by *domain names*. A domain name represents a path in a hierarchical tree structure, which in turn constitutes a *domain name space*. Each node of this tree is assigned a label, thus, a domain name is built as a sequence of labels separated by a dot, from a particular node up to the root of the tree.

A distinguishing feature of the design of DNS is that the administration of the system can be distributed among several (authoritative) name servers. A *zone* is a contiguous part of the domain name space that is managed by a set of authoritative name servers. Then, distribution is achieved by delegating part of a zone administration to a set of delegated sub-zones. DNS is a widely used scalable system, but it was not conceived with security concerns in mind, as it was designed to be a public database with no intentions to restrict access to information. Nowadays, a large amount of distributed applications make use of domain names. Confidence on the working of those aplications depends critically on the use of trusted data: fake information inside the system has been shown to lead to unexpected and potentially dangerous problems.

Already in the early 90's serious security flaws were discovered by Bellovin and eventually reported in [Bel95]. Different types of security issues concerning the working of DNS have been discussed in the literature, see, for instance, [Bel89, Bel95, Cer01, Gav93, CS94, Vix95]. Identified vulnerabilities of DNS make it possible to launch different kinds of attack, namely: *cache poisoning*, *client flooding*, *dynamic update vulnerability*, *information leakage* and *compromise of the DNS servers authoritative database* [Dav99, Bel95, AA04].

DNSSEC [AAL$^+$05a, AAL$^+$05c, AAL$^+$05b] is a suite of Internet Engineering Task Force (IETF) specifications for securing information provided by DNS. More specifically, this suite specifies a set of extensions to DNS which are oriented to provide mechanisms that support authentication and integrity of DNS data but not its vailability or confidentiality. In particular, the security extensions were designed to protect resolvers from forged DNS data, such as the one generated by DNS cache poisoning, by digitally signing DNS data using public-key cryptography. The keys used to sign the information are authenticated via a chain of trust, starting with a set of verified public keys that belong to the DNS root zone, which is the trusted third party.

The DNSSEC standards were finally released in 2005 and a number of testbeds, pilot deployments, and services have been rolled out in the last few years [Fou11, IAN11, IRL11, Nom11, PIR11, ABD$^+$, YOM$^+$11]. In particular, the main objective of the OpenDNSSEC project [ABD$^+$] is to develop an open source software that manages the security of domain names on the Internet.

Contributions

The important and critical role of DNSSEC in software systems and networks makes it a prime target for formal verification. We are not aware of projects that have set out to formally verify the correctness of DNSSEC implementations. Reasoning about implementations provides the ultimate guarantee that the deployment of these protocols satisfies the expected properties. We are convinced, though, of the need for complementary approaches where verification is performed on simplified models that abstract away from the specifics of any particular implementation.

This article initiates such an approach by developing a minimalistic specification of a DNSSEC model, and yet provides a realistic setting in which to explore the security issues that pertain to the realm of DNS. The specification puts forward an abstract formulation of the behavior of the protocol and the corresponding security-related events, where security goals, such as the prevention of cache poisoning attacks, can be given a formal treatment. In particular the formal model provides the grounds needed to formally state and verify security properties concerning the chain of trust generated along the DNSSEC tree and the prevention of cache poisoning attacks.

The specification has been developed using the Calculus of Inductive Constructions (CIC) [CH88, CP90, PM93] and it was formally verified using the Coq proof assistant [The10, BC04]. The Coq system is a free open source software that provides a (dependently typed) functional programming language and a reasoning framework based on higher order logic to perform proofs of programs. As one example of its applicability, Coq has been used for the development and formal verification of a compiler of a large subset of the C programming language [Ler09].

A detailed description of the specification of the model is presented in Spanish in [Baz11]. That document and the Coq development are available in http://www.fing.edu.uy/inco/grupos/gsi/sources/dnssec.

Contents of the paper

The rest of the paper is organized as follows: Section 2 provides a primer on DNS and DNSSEC focusing on the elements that are most relevant for our formal model, which we develop in Section 3. Section 4 presents some of the verified properties and outlines the corresponding proofs. Finally, Section 5 concludes and discusses directions for future work.

2 A Primer on the Vulnerabilities of DNS

In this section we provide some background on DNS, its vulnerabilities and the security extensions specified in DNSSEC.

The administration of DNS can be distributed among several (authoritative) name servers. DNS defines two types of server:

- *Nameservers*: which are authoritative servers that manage data of a contiguous part of the domain name space and where the master files reside. For redundancy sake, primary and secondary nameservers are provided for each zone.

- *Resolvers*: which are standard programs used to interact with the nameservers to extract information in response to client requests.

The process of obtaining information from DNS is called *name resolution*. Each server is initialized with the contact information of some authoritative servers of the root zone. Moreover, the root servers know how to contact the authoritative name servers of second level (e.g. the domains com, net, edu). Second level servers know the information of third level servers, and so on. By these means, the requestor can proceed refining the sought response by walking the tree structure, contacting different servers and getting “closer” to the answer after each referral.

Servers store the results of previous queries in their caches in order to speed up the resolution process, but as the mapping of names evolves, cached data has a limited time of validity. Authoritative servers attach, for instance, a Time To Live atribute (TTL) to this stored data, indicating when it should be removed from the cache. For more details concerning the working of DNS we refer to [LA06, Moc87a, Moc87b].

In an early work, Cheung and Levitt [CL00] discuss security issues of DNS and provide formal basis to model and prove security mechanisms that can be added to the system to prevent two specific problems:

- *Failure to authenticate DNS responses*: The message authentication mechanism used by DNS is weak. DNS checks if a received response matches a previously asked query only by checking if the Id attached in the query’s header matches with the Id of the pointed response. In this way, if an attacker can predict the used query Id and his answer reaches before the real one does (as if a name server receives multiple responses for its query, it uses the first one), it will be able to send a forged response.

- *Cache poisoning attacks*: An attacker can take advantage of the lack of authentication mechanisms to intentionally formulating misleading information, injecting bogus information into some server DNS cache. Having cached this information, the cheated DNS server is likely to get a Denial of Service (DoS), if the attacker sends a negative response that could actually be resolved; or in the worst case, the attacker can masquerade as a trusted entity, and then be able to intercept, analyze and intentionally corrupt the communication.

Cache poisoning attacks exploits a flaw in DNS, namely, the weak mechanisms used to ensure the authentication of data origin. As one of the goals of the DNS security extensions was to solve these vulnerabilities, we have put special focus in developing a formal specification that allows us to reason on the effectiveness of those extensiond regarding impersonation and cache poisoning attacks.

The extensions introduced by DNSSEC to improve the security of DNS require the combined application of mechanisms that make it possible to i) sign data, using public key cryptography, within zones ii) generate a chain of trust along the DNSSEC tree iii) perform key exchange within parent-child zones, and regular key rollover routines. The security extensions provide origin authentication and integrity assurance services for DNS data, including mechanisms for public key distribution and authenticated denial of existence of DNS data. However, respecting the principle assumed in the design of DNS that all data in the system must be visible, DNSSEC is not designed to provide confidentiality [AAL+05a].

The specification of the security extensions does not prevent the interaction of secure name servers and resolvers with non-secure ones. However, any communication that involves an unsecure server results in the loss of all DNSSEC security related capabilities. For this reason, in our model it is assumed that every server is security aware.

3 Formalization of the DNSSEC Model

In this section, we present an abstract model of DNSSEC. First, we introduce auxiliary definitions, to proceed then to define the set of states and a notion of valid state. Finally, we define the semantics of security-related events as state transformers and provide a formal definition of their execution. We start by providing notation used in this document.

3.1 Notation

We use standard notation for equality and logical connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\forall$, $\exists$). We extensively use record types and enumerated types. Record types definitions are of the form $R \stackrel{\text{def}}{=} [\![\, field_0 : A_0, \ldots, field_n : A_n \,]\!]$. Field selection is abbreviated using dot notation. We define an inductive relation I by giving introduction rules of the form:

$$rule \ \frac{a_0 \ldots a_j}{I\ b_0 \ldots b_n}$$

where free occurrences of variables are implicitly universally quantified.

Sets of type A are defined as *set* A, where *set* is an inductive type encoding sets as lists. We use $\{a_0, \ldots, a_n\}_A$ to denote the set of elements $a_0, \ldots, a_n$ of type A. When the type A is obvious from the context, we write $\{a_0, \ldots, a_n\}$. Classical notation is used for set operations ($\cup, \cap, \setminus, \in, \subseteq$).

3.2 Formalizing States

The state of a DNSSEC system consists of a collection of components that we now proceed to describe. Servers distributed globally are represented as objects of an abstract type **Process**.

3.2.1 Secure Resource Records

A Resource Record (*RR*) is the basic unit of information used in DNS. A structure of this kind is defined by six fields: i) the field *NAME* defines the domain name that applies to the given RR, ii) the field *TYPE* indicates the type of resource record. Figure 1 depicts the most common ones, iii) the field *CLASS* is used to identify the protocol group to which the record belongs. In the sequel we shall only make use of the class IN, which is the one of interest to people using TCP/IP software, since it stands for *Internet*, iv) *TTL* stands for Time To Live; it is primarily used by resolvers, and specifies how long a resource record should be cached before discarding it, v) the field *RDLENGTH* is an unsigned integer that specifies the length of the RDATA field, vi) the field *RDATA* contains the data of the resource record. The data field is defined differently for each type and class of data.

Type	*Description*
A	Internet Address
CNAME	Canonical Name (nickname pointer)
HINFO	Host Information
MX	Mail Exchanger
NS	Name Server
PTR	Pointer
SOA	Start Of Authority

Figure 1: Types of Resource Records

Formally, a resource record is defined as an object of the following record type:

$$\mathbf{RR} \stackrel{\text{def}}{=} [\![\quad \begin{aligned}&\mathit{Rdname}:\mathit{DName},\ \mathit{Rtype}:\mathit{RRType},\ \mathit{RClass}:\mathit{RRClass},\\ &\mathit{Rttl}:\mathit{TTL},\ \mathit{RDL}:\mathit{RDLength},\ \mathit{Rrdata}:\mathit{RData}\ ]\!]\end{aligned}$$

A set of resource records that share the same domain name, type and class is formalized as an object of the following type:

$$\mathbf{RRset} \stackrel{\text{def}}{=} \mathit{set}\ \mathit{RR}$$

To implement the proposed security extensions, DNSSEC introduces additional security-related resource records: i) for origin authentication DNSSEC provides a hierarchical public key infrastructure (PKI), which allows resolvers to obtain the DNSSEC key of a zone and use it for authenticating signed data belonging to that zone. To support this PKI, three resource records were introduced: RRSIG, DNSKEY and DS, ii) for assuring integrity of data each zone signs all the RRsets over it is authoritative. In every transmission, RRSIGs are transmitted along with the replied RRsets, and by these means, when a transmission is received, data integrity can be verified, iii) for authenticated denial of existence, Next Secure (NSEC) resource records are provided. They list all of the existent RRs belonging to an owner name within an authoritative zone, making it possible to verify the non-existence of a RR, by comparing against the RR list of its owner name. Figure 2 describes the security oriented new resource records.

RR	**Description**
RRSIG	Signature over RRset made using private key
DNSKEY	Public key, needed for verifying a RRSIG
DS	Delegation Signer, pointer for building chains of trust. The Parent DNSKEY signs the Parent DS, Parent DS signs the Child DNSKEY, and so forth, providing a mechanism to verify origin integrity from a domain name up to the root servers
NSEC	Used to provide an authenticated non-existence of data. Indicates which name is the next one in the zone and which type codes are available for the current name

Figure 2: Resource records introduced by DNSSEC

The type of a resource record, which includes the specific types defined by the DNSSEC specifications, is defined by the following enumerated type:

$$\begin{aligned}\textbf{RRType} \overset{\text{def}}{=}\ & A \mid PTR \mid NS \mid CNAME \mid MX \mid SOA \mid HINFO \\ & \mid \textbf{RRSIG} \mid \textbf{DNSKEY} \mid \textbf{DS} \mid \textbf{NSEC}\end{aligned}$$

We formalize the notion of secure (set of) RR by the following two types:

$$\begin{aligned}\textbf{SecRR} &\overset{\text{def}}{=} [\![\, RR' : RR,\ Rsign : RR \,]\!] \\ \textbf{SecRRset} &\overset{\text{def}}{=} [\![\, RRset' : RRset,\ RRsign : RR \,]\!]\end{aligned}$$

where *Rsign* and *RRsign* contain, respectively, the signature corresponding to RR' and to $RRset'$.

3.2.2 The Distributed Database

DNS manages a distributes database, which is indexed by a tuple (dname, type, class) of type *Idx*:

$$\textbf{Idx} \overset{\text{def}}{=} [\![\, Idname : DName,\ Itype : RRType,\ IClass : RRClass \,]\!]$$

The range of this database is a set of secure RRs. A DNS database is thus formalzed as an object of the function type:

$$\textbf{DbMap} \overset{\text{def}}{=} Idx \rightarrow SecRRset$$

3.2.3 DNS Message

A DNS(SEC) message consists of a header and four additional sections, as illustrated in Figure 3. The *HEADER* contains an identification (Id) field, which is used to match an answer to its corresponding query. The *QUESTION* section consists of a target domain name (*QNAME*),

HEADER	QUESTION	ANSWER	AUTHORITY	ADDITIONAL

Figure 3: DNS Message Format [Dav99]

a type (*TYPE*), and a class (*QCLASS*). A query to find out, for instance, the IP address of the host *h1.fceia.unr.edu.ar* will have *QNAME=h1.fceia.unr.edu.ar*, *QTYPE=A* and *QCLASS=IN*. The *ANSWER* section contains RRs which directly answer the query. The *AUTHORITY* section carries RRs which describe other authoritative server (e.g. may contain RRs to refer the querier to other name servers during the resolution process). The *ADDITIONAL* section carries RRs that may be helpful for using RRs information of other sections (e.g. may contain **A** RRs to provide the IP address for the RRs listed in the authority section).

We define the *DNSMessage* type as follows:

$$\mathbf{DNSMessage} \overset{\text{def}}{=} \llbracket \begin{array}{l} Hdr\!:\!Header,\ Q\!:\!Idx,\ Ans\!:\!SecRRset, \\ Auth\!:\!SecRRset,\ Additional\!:\!SecRRset \ \rrbracket \end{array}$$

3.2.4 Pending Submissions

Pending submissions, that is to say, requests or answers, possibly triggered by a received answer, which are waiting to be delivered, are defined as objects of the following type:

$$\mathbf{SendQR} \overset{\text{def}}{=} \llbracket\, SendQ\!:\!set\ InfoMsg,\ SendR\!:\!set\ InfoMsg \,\rrbracket$$

An object of the type $InfoMsg$ represents the required information needed to be sent.

$$\mathbf{InfoMsg} \overset{\text{def}}{=} \llbracket\, MFrom\!:\!Process,\ MTo\!:\!Process,\ MMsg\!:\!DNSMessage \,\rrbracket$$

3.2.5 System Keys

DNSSEC security is based in the use of public key cryptography. Each DNS server owns two keys, namely, a Zone Signing Key (ZSK) and a Key Signing Key (KSK). A ZSK key is used to sign every RRset within a zone, generating the RRSIG records. A KSK key is used specifically to sign the DNSKEY RRset and generate its corresponding RRSIG. To access the system's DNSKEYS, the following record have been defined:

$$\mathbf{keySet} \overset{\text{def}}{=} \llbracket\, key\!:\!RR,\ ksign\!:\!RR \,\rrbracket$$

where *key* is a DNSKEY and *ksign* its corresponding RRSIG signature.

Each server has its corresponding pair of keys:

$$\mathbf{keys} \overset{\text{def}}{=} \llbracket\, zsk\!:\!keySet,\ ksk\!:\!keySet \,\rrbracket$$

3.2.6 Delegations

A delegation describes a father-child relationship between DNSSEC servers. As shown in Figure 2, it helps building the DNSSEC chain of trust. To model delegation in our specification we have defined the following function:

$$\mathbf{DSpDB} \stackrel{\text{def}}{=} Process \rightarrow DSp$$

where:

$$\mathbf{DSp} \stackrel{\text{def}}{=} [\![\, Tsrv : Process,\ rrDS : SecRR \,]\!]$$

In words, we have defined a function that maps Parent servers with its corresponding Childs to whom they trust, along with the digest RR to prove it. This is an essential part of the model, as it will allow us to reason over the effectiveness of the chain of trust.

3.2.7 System State

To reason about the DNSSEC security system most details of the state may be abstracted. States are modeled by a record type with nine components:

$$\begin{aligned}\mathbf{State} \stackrel{\text{def}}{=} [\![\, & Servers : set\ Process,\\ & TrustedServers : set\ Process,\\ & ServerKeys : Process \rightarrow keys,\\ & Delegations : Process \rightarrow DSpDB,\\ & Parents : Process \rightarrow DSp,\\ & viewAuth : Process \rightarrow DbMap,\\ & viewCache : Process \rightarrow DbMap,\\ & PendingQueries : Process \rightarrow set\ Header,\\ & SendBuffer : Process \rightarrow SendQR \,]\!]\end{aligned}$$

The component *Servers* indicates the involved DNS servers, whereas that *TrustedServers* and *ServerKeys* represent set of publicly known trusted servers and the set of keys for every server, respectively. The delegations issued by each server and the fathers of a server are indicated by the components *Delegations* and *Parents*. The components *viewAuth* and *viewCache* are used to represent the authoritative view and the cache view of every server, *PendingQueries* is a function that maps a server with the expected answers to the already performed queries, and *SendBuffer* is a buffer with the corresponding pending submissions for each server.

3.2.8 Valid State

We define a notion of valid state that captures essential security properties of the system, and more particularly of the DNSSEC specification provided by the "DNSSEC protocol document set", which refers to the three documents that form the core of the DNS security extensions:

"DNS Security Introduction and Requirements" [AAL+05a], *"Resource Records for DNS Security Extensions"* [AAL+05c], and *"Protocol Modifications for the DNS Security Extensions"* [AAL+05b].

We say that the predicate $Valid$ holds on state s if s satisfies the following properties:

1. *"Every RRset within the view of a server must be signed by its corresponding RRSIG signature"*, which can be stated formally as follows,

$$\begin{aligned}\textbf{RR\_integrity}\ s \overset{\text{def}}{=} & \ \forall\ srv\!:\!Process,\ srv \in s.Servers \to \\ & signedView\ s\ srv\ (s.viewAuth\ srv) \land \\ & signedView\ s\ srv\ (s.viewCache\ srv)\end{aligned}$$

   where $signedView$ verifies that every RRset of a sever's zone is correctly signed by its corresponding RRSIG.

2. *"Every server's Zone keys must be signed by its corresponding Key Signing Key (KSK)"*,

$$\begin{aligned}\textbf{ZSK\_integrity}\ s \overset{\text{def}}{=} & \ \forall\ (srv\!:\!Process),\ srv \in s.Servers \to \\ & verifySign\ (s.ServerKeys\ srv).ksk.key \\ & \quad \{(s.ServerKeys\ srv).ksk.key\} \cup \{(s.ServerKeys\ srv).zsk.key\} \\ & \quad (s.ServerKeys\ srv).zsk.ksign\end{aligned}$$

   where $verifySign$ checks that the RRSIG of a RRset has been effectively generated from a given key.

3. *"Every server must be publicly known as trusted, or be verified by the corresponding digest in its father's zone"*,

$$\begin{aligned}\textbf{KSK\_integrity}\ s \overset{\text{def}}{=} & \ \forall\ (srv\!:\!Process),\ srvH \in s.TrustedServers \lor \\ & checkDigest\ s\ srvH\ (s.Parents\ srvH).rrDS.RR'\end{aligned}$$

   where $checkDigest$ validates if a DS record in the Parent zone really contains a digest of its Child keys.

We thus formally define the $Valid$ predicate over $State$ as the conjunction of the previous validity conditions:

$$\textbf{Valid}\ s \overset{\text{def}}{=} RR\_integrity\ s \land ZSK\_integrity\ s \land KSK\_integrity\ s$$

3.3 Events

The working of the system is modelled in terms of the execution of events. An event is understood as an action that transforms the state of the system. Next we present the syntax, and specify the semantics and execution of events.

Table 1: Events

Name	Description
Add_Server	Creates a new server in the domain name system
Delete_Server	Deletes a server from the domain name system
Add_RRset	Inserts a new RRset in the authoritative view of a given server
Delete_RRset	Deletes a RRset from the authoritative view of a given server
Server_ZSK_rollover	Performs a rollover of the zone key of a given server
Server_KSK_rollover	Performs a rollover of the KSK of a given server
Add_TrustedServer	Indicates the reliable publication of a server known as trusted
Del_TrustedServer	Indicates that a server should no longer be consider as trusted
Send_Query	Sends a specific query to a given server
Receive_Response	Receives a response from a given server
Send_PendingQ	Sends pending queries
Send_PendingR	Sends pending responses
RR_TimeOut	Indicates that the TTL of a given RRset has expired

3.3.1 Syntax

The syntax and signature for every event relevant to our abstract model of DNSSEC is formalized by the non-recursive inductive set *Event*:

$$
\begin{array}{lll}
\textbf{\textit{Event}} & \stackrel{\text{def}}{=} & \\
& | & Add\_Server : Process \rightarrow keys \rightarrow DSpDB \rightarrow DSp \rightarrow DbMap \rightarrow DbMap \rightarrow Event \\
& | & Delete\_Server : Process \rightarrow Event \\
& | & Add\_RRset : Process \rightarrow Idx \rightarrow RRset \rightarrow Event \\
& | & Delete\_RRset : Process \rightarrow Idx \rightarrow Event \\
& | & Server\_ZSK\_rollover : Process \rightarrow RR \rightarrow Event \\
& | & Server\_KSK\_rollover : Process \rightarrow RR \rightarrow RR \rightarrow Event \\
& | & Add\_TrustedServer : Process \rightarrow Event \\
& | & Del\_TrustedServer : Process \rightarrow Event \\
& | & Send\_Query : Process \rightarrow Process \rightarrow DNSMessage \rightarrow Event \\
& | & Receive\_Query : Process \rightarrow Process \rightarrow DNSMessage \rightarrow Event \\
& | & Receive\_Response : Process \rightarrow Process \rightarrow DNSMessage \rightarrow Event \\
& | & Send\_PendingQ : Process \rightarrow Event \\
& | & Send\_PendingR : Process \rightarrow Event \\
& | & RR\_TimeOut : Process \rightarrow Idx \rightarrow RR \rightarrow Event
\end{array}
$$

A brief description of each event is shown in Table 1. In the next section we present the formal semantics for a small subset of these events.

3.3.2 Semantics

The behavior of the events is specified by their pre- and postconditions, which are given by the predicates *Pre* and *Post* respectively,

$$\begin{aligned}&Pre : State \rightarrow Event \rightarrow Prop\\&Post : State \rightarrow State \rightarrow Event \rightarrow Prop\end{aligned}$$

Preconditions are defined in terms of the system state while postconditions are defined in terms of the before and after states. Due to space constraints, we only present here the formal specification of three distinguished events, namely, *Server_ZSK_rollover*, *Receive_Response* and *RR_TimeOut*. In the specification of the postconditions we have omitted all fields of the state that remain invariant when executing the corresponding event. The names of the auxiliary functions and predicates used should be self-explanatory.

- ***Server_ZSK_rollover srv rrzsk*** This event performs the rollover of the ZSK key for a specific server *srv*. For this event to take place it is needed that *srv* exists in the system and *rrzsk* should be a ZSK key, that is to say,a key which is identified as a *Zone Signing key* by the information of its RRDATA.

$$\textbf{\textit{Pre s (Server\_ZSK\_rollover srv rrzsk)}} \stackrel{\text{def}}{=}$$
$$isServer\ s\ srv\ \wedge\ isZSK\ rrzsk$$

$$\textbf{\textit{Pos s s' (Server\_ZSK\_rollover srv rrzsk)}} \stackrel{\text{def}}{=}$$
$$\begin{aligned}&(s'.ServerKeys\ srv).zsk.key = rrzsk\\&\wedge\ (s'.ServerKeys\ srv).zsk.ksign =\\&\quad sign\ (s.ServerKeys\ srv).ksk.key\ (\{(s.ServerKeys\ srv).ksk.key\} \cup \{rrzsk\})\\&\wedge\ \forall\ (i{:}Idx),\ (s'.viewAuth\ srv\ i).RRsign = sign\ rrzsk\ (s.viewAuth\ srv\ i).RRset'\end{aligned}$$

  The postcondition states that when this event executes successfully the key *rrzsk* is stored as the new ZSK, its signature is calculated, using *sign*, and stored as its corresponding RRSIG. In addition to that, all the signatures for the resource records within the authoritative view of the server *srv* are re-calculated.

- ***Receive_Response srv_from srv_to m*** This event models the processing of an answer message *m*. For this operation to succeed, servers *srv_from* and *srv_to* must belong to the set of servers of the system and the received response should be an answer to a previously submitted query delivered by the server *srv_to*.

$$\textbf{\textit{Pre s (Receive\_Response srv\_from srv\_to m)}} \stackrel{\text{def}}{=}$$
$$\begin{aligned}&isServer\ s\ srv\_from\ \wedge\ isServer\ s\ srv\_to\\&\wedge\ m.Hdr \in (s.PendingQueries\ srv\_to)\end{aligned}$$

Pos s s' (Receive_Response srv_from srv_to m) $\stackrel{\text{def}}{=}$

```
$if\ (nodata\_newrefer\ m)$
    $(s'.SendBuffer\ srv\_to).SendQ = (s.SendBuffer\ srv\_to).SendQ\ \cup$
        $\{(makeResp\ srv\_to\ (next\ m.Additional.RRset')\ m)\}$
$elseif\ (verifySign\ (s.ServerKeys\ srv\_to).zsk.key\ m.Ans.RRset'\ m.Ans.RRsign)$
    $((s'.viewCache\ srv\_to)\ m.Q).RRset' = m.Ans.RRset'$
    $\wedge\ ((s'.viewCache\ srv\_to)\ m.Q).RRsign = m.Ans.RRsign$
    $\wedge\ s'.PendingQueries\ srv\_to = (s.PendingQueries\ srv\_to) \setminus \{m.Hdr\}$
```

The postconditions establishes the conditions required to be satisfied for a given resource record to be accepted and allocated in the cache view of a server. In the case that the received answer does not contain information in its auth section, but it does contain a *closer* server to refer, then the message will be re-sent to that server. On the contrary, if the received message contains data in its auth section, and the received RRsets as well as their corresponding received signatures are verified by the sender zsk key, then the RRsets and their signatures are added to the server's cache view and, as the query has been successfully replied, the message is removed from the *PendingQueries* set. It should be noticed that a correct execution of this event would prevent a cache poisoning like the one discussed in Section 2. In section 4 we will skecth the proof that the state remains valid after the execution of this event.

- ***RR_TimeOut srv i rr*** Runs on the expiration of the TTL of a resource record *rr* for a given server *srv*. The operation precondition will be verified when *srv* belongs to the set of servers of the system and the TTL of *rr* has expired due to timeout of either the resource record or its corresponding signature.

Pre s (RR_TimeOut srv i rr) $\stackrel{\text{def}}{=}$

```
$isServer\ s\ srv\ \wedge\ isTimeout\ s\ srv\ i\ rr$
$\wedge\ (rr\ \in\ (s.viewCache\ srv\ i).RRset'\ \vee\ rr\ \in (s.viewAuth\ srv\ i).RRset')$
```

Pos s s' (RR_TimeOut srv i rr) $\stackrel{\text{def}}{=}$

```
$if\ (rr\ \in (s.viewAuth\ srv\ i).RRset')$
    $((s'.viewAuth\ srv)\ i).RRset' = ((s.viewAuth\ srv)\ i).RRset' \setminus \{rr\}$
    $\wedge\ ((s'.viewAuth\ srv)\ i).RRsign =$
        $sign\ (s.ServerKeys\ srv).zsk.key\ ((s'.viewAuth\ srv)\ i).RRset'$
$else\ ((s'.viewCache\ srv)\ i).RRset' = ((s.viewCache\ srv)\ i).RRset'\{rr\}$
    $\wedge\ ((s'.viewCache\ srv)\ i).RRsign =$
        $sign\ (s.ServerKeys\ srv).zsk.key\ ((s'.viewCache\ srv)\ i).RRset'$
```

The postcondition of *RR_TimeOut* states that the expired resource record and its signature are removed from the correponding authoritative or cache view.

3.3.3 Errors

When an event is executed and a precondition is not valid, the system answers with a corresponding error code. These error codes are defined by the enumerated type $ErrorCode$:

$$\begin{aligned}\textbf{\textit{ErrorCode}} \overset{\text{def}}{=}\ & server\_not\_exists \mid invalid\_zsk \mid query\_not\_asked \\ & \mid rrset\_not\_exists \mid rr\_not\_timeout \mid \ldots\end{aligned}$$

Table 2 specifies the error code associated to unverified preconditions of the three events whose semantics was presented in Section 3.3.2. In our model this is formalized as a relation $ErrorMsg$ between valid states of the system and error messages.

Table 2: Error messages

Server_ZSK_rollover srv rrzsk	
$\neg\ isServer\ s\ srv$	server_not_exists
$\neg\ isZSK\ rrzsk$	invalid_zsk
Receive_Response srv_from srv_to m	
$\neg\ isServer\ s\ srv\_from \vee\ \neg\ isServer\ s\ srv\_to$	server_not_exists
$\neg\ m.Hdr\ \in\ s.PendingQueries\ srv\_to$	query_not_asked
RR_TimeOut srv i rr	
$\neg\ isServer\ s\ srv$	server_not_exists
$rr\ \notin\ (s.viewCache\ srv\ i).RRset'$ $\vee\ rr\ \notin\ (s.viewAuth\ srv\ i).RRset'$	rrset_not_exists
$\neg isTimeout\ s\ srv\ i\ rr$	rr_not_timeout

3.3.4 One-step Execution

Executing an event e over a state s produces a new state s' and a corresponding answer r (denoted $s \xhookrightarrow{e/ans} s'$), where the relation between the former state and the new one is given by the postcondition relationship $Post$.

$$\textbf{\textit{exec\_pre}}\ \frac{Pre\ s\ e \qquad Post\ s\ s'\ e}{s \xhookrightarrow{e/ok} s'}$$

$$\textbf{\textit{exec\_npre}}\ \frac{\neg\ Pre\ s\ e \qquad \exists\ ec\!:\!ErrorCode,\ ErrorMsg\ s\ e\ ec \wedge ans = error\ ec}{s \xhookrightarrow{e/ans} s}$$

If the precondition $Pre\ s\ e$ is satisfied, then the resulting state s' and the corresponding answer ans are the ones described by the relation $exec$. However, if $Pre\ s\ e$ is not satisfied, then the state s remains unchanged and the system answer is the error message determined by the relation

$ErrorMsg$. Formally, the possible answers of the system are defined by the type *answer*:

$$\begin{array}{rcl} \textbf{\textit{answer}} & \stackrel{\text{def}}{=} & ok : answer \\ & | & error : ErrorCode \rightarrow answer \end{array}$$

where *ok* is the answer resulting from a successful execution of an event.

4 Verification of Security Properties

In this section we discuss two relevant properties of the model that have been formally stated and verified. We first concentrate on the proof that one-step execution preserves the validity of states. We sketch the proof of this property and show that the notion of valid state embodies necessary conditions to prove the objective security properties. Then, we show, formally, that to have this invariance result is not enough to ensure consistency of the chain of trust.

4.1 An Invariant of One-step Execution

A one-step execution invariant is a property that if it holds for the state before the execution of any event it remains valid for the state resulting from that execution. We show that the validity of the model state, as defined in Section 3.2, is a one-step invariant of our specification.

Proposition 1 *For any $s\ s' : State$, $ans : answer$ and $e : Event$, if Valid s and $s \xhookrightarrow{e/ans} s'$ hold, then Valid s' also holds.*

Proof. The proof of this proposition proceeds by case analysis on the execution $s \xhookrightarrow{e/ans} s'$: i) if the precondition $Pre\ s\ e$ is not satisfied, then the specification of executions establishes that the state s is not modified and that $s = s'$, so s' is trivially valid because s is valid, ii) otherwise, $Post\ s\ s'\ r\ e$ must hold and then we proceed by case analysis on the event e.

We shall here discuss in detail the proof argument for the case the event e is of the form $Receive\_Response\ srv\_from\ srv\_to\ m$. The complete formal proof of this and each of the remaining cases can be found in the accompanying formalization.

The proof start by analyzing the type of response obtained. In the case the answer has authoritative content we proceed by checking whether the signature of the received RRset is verified. Then we have two possible cases:

1. If the received answer does not contain data, then, as specified in its postconditions, when executing *Receive_Response*, the component *viewCache* of the state s' will remain invariant, moreover the only component modified in s' will be $SendBuffer$ which is not verified in the validity property, so we can state that s' remain valid.

2. On the contrary, if the answer contains data in its $Auth$ section, we must analyze if the received *RRset* is verified by its corresponding signature *RRsig*: i) if the signature is not verified, the received answer will be discarded and no component of the state will be modified, so the property will remain satisfied for s', ii) if the signature is correctly verified and so the *RRset* is added to the *viewCache* view of the server srv_to, the resultant state

s' will stay valid, as *viewCache* is the only modified component and, by adding verified RRset and RR signature, it remains correctly signed.

□

Observe that the instantiation of Proposition 1 to the case in which the executed event is *Receive_Response* reflects the (informal) security requirement that no man-in-the-middle can masquerade as a trusted entity in order to provide answers with fake resource records. By proving the correct execution of this event, we are providing (formally verified) evidence that if a classical DNS attacker sends a malicious resource record to a secure server that record shall be discarded as it will not be verified by its corresponding signature, which, in turn, has been created by its father within the chain of trust. This is clearly a security improvement regarding DNS, because the reception mechanism of this system is what makes it a potential victim of cache poisoning attacks.

4.2 Compromising the Chain of Trust

It is important to notice that the conditions specified for a state of our model to be a valid one constitute an almost straightforward interpretation of the (security) recommendations laid down in relevant DNSSEC RFCs like, for instance, [AAL+05a, AAL+05c, AAL+05b]. We can show, however, that despite the validity of states is preserved by execution this does not necessarily guarantee that the chain of trust remains valid. In particular, analyzing the conditions required for the rollover of a zone key, which in our model is specified as part of the semantics of the event *Server_ZSK_rollover*, we have detected a small inconsistency concerning the data of the system. Namely, for a resource record to be discarded from a view either authoritative or cache, it is only required that its corresponding TTL is recached. Now, a rollover of a zone key might be needed to be executed, for instance, in the case the server's zone is compromised, even if the TTL of the key has not expired. Consequently, every RRset within the zone must be signed to generate the new RRSIG records. Therefore, every DNS server that contains these, just re-signed, records inside its cache view will become inconsistent. This issue could not be detected during the verification of the invariance of the event *Server_ZSK_rollover* because the specification of the DNSSEC protocol mandates for a resource record to be discarded from the zone file only in if its TTL has expired.

This problem was independently discovered and pointed out by Bau and Mitchell in [BM10], where they study the security properties of a restricted model of DNSSEC using model checking techniques. We adhere to their recommendation that the resolver logic specified in the corresponding RFC [AAL+05a] should be strengthened so as to prevent the identified problem.

We now proceed to provide a formal argument that the (incomplete) current specification of the zone key rollover procedure may facilitate the occurrence of inconsistent chains of trust. DNSSEC provides no mechanisms to validate a cached resource record faced to another signature in its attestation chain. We shall show that if for some reason it is performed a rollover of the zone key for a given server (*srv*) which has not yet arrived to its TTL a resolver (*srvOldCache*) caching the corresponding resource record (identified by its index a) signed with the old zone key may keep cached that resource record for the entire signature validity period, turning the previous valid chain of trust into an inconsistent one.

We assume that the server *srvOldCache* has information in its cache view corresponding to the authoritative zone of the server *srv*, i.e. for an index *a*:

$$((s.viewAuth\ srv)\ a).RRsign = ((s.viewCache\ srvOldCache)\ a).RRsign \tag{1}$$

Now, let us consider a correct execution of the event *Server_ZSK_rollover*. Thus, for arbitrary $s' : State$ and $rrzsk : RR$, the following predicates are verified:

$$\begin{aligned} &Valid\ s \\ &Pre\ s\ (Server\_ZSK\_rollover\ srv\ rrzsk) \\ &Post\ s\ s'\ (Server\_ZSK\_rollover\ srv\ rrzsk) \end{aligned}$$

Therefore, when performing a rollover of the zone key zsk for the server *srv*, the following inconsistency will take place:

$$((s'.viewAuth\ srv)\ a).RRsign \neq ((s'.viewCache\ srvOldCache)\ a).RRsign \tag{2}$$

The proof will be performed by reduction to the absurd, assuming that the mentioned inconsistency in fact does not occur, considering as valid:

$$((s'.viewAuth\ srv)\ a).RRsign = ((s'.viewCache\ srvOldCache)\ a).RRsign \tag{3}$$

We know from the postcondition of the event *Server_ZSK_rollover* that:

$$(s'.ServerKeys\ srv).zsk.key = rrzsk \tag{4}$$
$$\forall\, i{:}Idx,\ ((s'.viewAuth\ srv)\ i).RRsign = sign\ rrzsk\ (s.viewAuth\ srv\ i).RRset' \tag{5}$$
$$s'.viewCache = s.viewCache \tag{6}$$

Rewriting (6) and then (1) in (3) we have that:

$$((s'.viewAuth\ srv)\ a).RRsign = ((s.viewAuth\ srv)\ a).RRsign \tag{7}$$

Now, considering that if a given record RR is signed with two different zsk keys, different RRSIG records are obtained, we arrive to:

$$((s'.viewAuth\ srv)\ a).RRsign \neq ((s.viewAuth\ srv)\ a).RRsign \tag{8}$$

The propositions (7) and (8) reflects an absurd. This allow us to conclude that (2) holds.

5 Conclusion and Future Work

We have developed an abstract model of DNS that incorporates the security extensions defined by the DNSSEC specification suite. We have established and proved the security properties required to be satisfied by the operational behaviour of an implementation of this version of DNS, which is specified in our model by an abstract state and the events that represent the working of the system. In particular, this result provides a formal means to assess the effectiveness of a (correct) deployment of the security requirements specified by DNSSEC to prevent cache poisoning attacks.

In addition to that, we have identified an exploitable vulnerability that, according to our understanding, emerges as a flaw of the specification in question. As shown in Section 4, the conditions that must be verified in the case a rollover of a zone key must be performed, as specified in the RFC 4033 [AAL+05a], do not suffice to ensure the validity of the chain of trust. We have sketched in this article a formal proof that shows how the chain of trust can be compromised if only the expiration time of a key is considered as the cause to perform the rollover procedure. This property is established as a lemma in the formalization available in http://www.fing.edu.uy/inco/grupos/gsi/sources/dnssec.

The formal development is about 5kLOC of Coq (see Figure 4), including proofs, and forms a suitable basis for reasoning about DNSSEC.

There are several directions for future work. One immediate direction is to extend our specification and to establish results concerning the impact of introducing resource records of type *NSEC*. An NSEC record points to the next valid name in the zone file and is used to provide proof of non-existence of names within a zone. Through repeated queries that return NSEC records it is possible to retrieve all of the names in the zone, a process commonly called *walking* the zone. This side effect of the NSEC architecture subverts policies frequently implemented by zone owners which forbid zone transfers by arbitrary clients. We see as an interesting and challenging task to specify ways of preventing zone walking by constructing NSEC records that cover fewer names [WI06].

Model and basic lemmas	2k
Valid state invariance	3k
Proof of inconsistency	0.2k
Total	5.2k

Figure 4: LOC of Coq development

We also intend to obtain an executable specification of the model in a high level functional language. This prototype shall be constructed as an algorithm that verifies the formal specification that has been developed and it shall be codified also using the proof assistant Coq. Once verified, it could be derived, using the program extraction mechanism of Coq, a high-level and correct implementation of the specified system.

Bibliography

[AA04] D. Atkins, R. Austein. Threat analysis of the domain name system. In *DNS). RFC 3833, Internet Engineering Task Force*. 2004.

[AAL+05a] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), Mar. 2005. Updated by RFC 6014.
http://www.ietf.org/rfc/rfc4033.txt

[AAL+05b] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035 (Proposed Standard), Mar. 2005. Updated

by RFCs 4470, 6014.
http://www.ietf.org/rfc/rfc4035.txt

[AAL+05c] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose. Resource Records for the DNS Security Extensions. RFC 4034 (Proposed Standard), Mar. 2005. Updated by RFCs 4470, 6014.
http://www.ietf.org/rfc/rfc4034.txt

[ABD+] R. Arends, R. Bellgrim, A. Dalitz, J. A. Dickinson, J. Jansen, S. Lloyd, M. Mekking, S. Morris, R. Post, Y. Schaeffer, J. Schlyter, P. Wallstrm. The OpenDNSSEC project.
http://www.test.org/doe/

[Baz11] E. Bazan. Especificación formal del modelo DNSSEC en el cálculo de construcciones inductivas. Master's thesis, FCEIA, Universidad Nacional de Rosario, Argentina, Oct. 2011.

[BC04] Y. Bertot, P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.
http://www.labri.fr/publications/l3a/2004/BC04

[Bel89] S. M. Bellovin. Security problems in the TCP/IP protocol suite. *SIGCOMM Comput. Commun. Rev.* 19:32–48, April 1989.
doi:http://doi.acm.org/10.1145/378444.378449
http://doi.acm.org/10.1145/378444.378449

[Bel95] S. M. Bellovin. Using the Domain Name System for System Break-ins. In *Proceedings of the 5th conference on USENIX UNIX Security Symposium - Volume 5*. Pp. 18–18. USENIX Association, Berkeley, CA, USA, 1995.
http://dl.acm.org/citation.cfm?id=1267591.1267609

[BM10] J. Bau, J. C. Mitchell. A Security Evaluation of DNSSEC with NSEC3. *IACR Cryptology ePrint Archive* 2010:115, 2010.

[Cer01] C. Cert/c. CERT Advisory CA-2001-02 Multiple Vulnerabilities in BIND. Aug. 2001.
http://www.cert.org/advisories/CA-2001-02.html

[CH88] T. Coquand, G. Huet. The Calculus of Constructions. In *Information and Computation*. Volume 76(2/3), pp. 95–120. Academic Press, Feb./Mar. 1988.

[CL00] S. Cheung, K. N. Levitt. A Formal-Specification Based Approach for Protecting the Domain Name System. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*. Pp. 25–28. 2000.

[CP90] T. Coquand, C. Paulin-Mohring. Inductively defined types. In Martin-Löf and Mints (eds.), *COLOG-88, International Conference on Computer Logic, Tallinn,* *USSR, December 1988*. Lecture Notes in Computer Science 417, pp. 50–66. Springer-Verlag, 1990.

[CS94] P. U. D. of Computer Sciences, C. Schuba. *Addressing weaknesses in the domain name system protocol*. CSD-TR / Computer Sciences Department, Purdue University n.º 28. Purdue University, Dept. of Computer Sciences, 1994.
http://books.google.com/books?id=QDHDPgAACAAJ

[Dav99] D. Davidowicz. Domain Name System (DNS) Security. 1999.
http://compsec101.antibozo.net/papers/dnssec/dnssec.html

[Fou11] T. I. I. Foundation. .se Top Level Domain. 2011.
http:/www.iis.se/

[Gav93] E. Gavron. RFC 1535: A Security Problem and Proposed Correction With Widely Deployed DNS Software. Oct. 1993. Status: INFORMATIONAL.
ftp://ftp.internic.net/rfc/rfc1535.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1535.txt

[IAN11] IANA. Interim Trust-Anchor Repository. 2011.
https:/itar.iana.org

[IRL11] U. C. D. Internet Research Lab. The SecSpider DNSSEC Monitoring Project. 2011.
http:/secspider.cs.ucla.edu/

[LA06] C. Liu, P. Albitz. *DNS and BIND*. Fifth edition, 2006.
http://www.oreilly.com/catalog/9780596100575

[Ler09] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM* 52:107–115, July 2009.
doi:http://doi.acm.org/10.1145/1538788.1538814
http://doi.acm.org/10.1145/1538788.1538814

[Moc87a] P. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.
http://www.ietf.org/rfc/rfc1034.txt

[Moc87b] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966.
http://www.ietf.org/rfc/rfc1035.txt

[Nom11] Nominet. Nominet DNSSEC Testbed. 2011.
http://www.nominet.org.uk/registrars/DNSSEC/

[PM93] C. Paulin-Mohring. Inductive Definitions in the system Coq - Rules and Properties. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*. Pp. 328–345. Springer-Verlag, London, UK, 1993.
http://dl.acm.org/citation.cfm?id=645891.671440

[PIR11] P. PIR. Org Top Level Domain. 2011.
http://www.iana.org/domains

[The10] The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.3. 2010.
http://coq.inria.fr

[Vix95] P. Vixie. DNS and BIND security issues. In *Proceedings of the 5th conference on USENIX UNIX Security Symposium - Volume 5*. Pp. 19–19. USENIX Association, Berkeley, CA, USA, 1995.
http://dl.acm.org/citation.cfm?id=1267591.1267610

[WI06] S. Weiler, J. Ihren. Minimally Covering NSEC Records and DNSSEC On-line Signing. RFC 4470 (Proposed Standard), Apr. 2006.
http://www.ietf.org/rfc/rfc4470.txt

[YOM$^+$11] H. Yang, E. Osterweil, D. Massey, S. Lu, L. Zhang. Deploying Cryptography in Internet-Scale Systems: A Case Study on DNSSEC. *IEEE Trans. Dependable Sec. Comput.* 8(5):656–669, 2011.