Especificación Formal del Modelo DNSSEC en el Cálculo de Construcciones Inductivas

Tesina de grado presentada

por

Ezequiel Bazán Eixarch

B3733/8

al

Departamento de Ciencias de la Computación

en cumplimiento parcial de los requerimientos

para la obtención del grado de

Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Universidad Nacional de Rosario

Av. Pellegrini 250, Rosario, República Argentina

Septiembre 2011

Supervisor

MsC. Carlos D. Luna

Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Julio Herrera y Reissig 565, Montevideo, Uruguay

Resumen

Con el crecimiento de aplicaciones desarrolladas en base al uso de nombres de dominio, la autenticidad en los datos dentro de DNS (Domain Name System) se ha tornado crítica, haciendo que información falsa dentro del sistema pueda llevar a problemas inesperados y potencialmente peligrosos. Para proveer extensiones de seguridad al protocolo DNS, la IETF (Internet Engineer Task Force) desarrolló DNSSEC (DNS SECurity Extensions).

El presente trabajo provee una especificación formal de DNSSEC, utilizando el Cálculo de Construcciones Inductivas (CCI) y COQ como asistente de pruebas. El enfoque propuesto aborda principalmente el análisis de integridad de la cadena de confianza que se genera a lo largo del árbol DNSSEC, así como también la posibilidad de que se produzca algún tipo de contaminación de caché por suplantación de datos.

La formalización en el CCI permitió realizar un análisis riguroso de la especificación propuesta. Se logró demostrar que la semántica de los comandos que se ejecutan dentro del sistema conservan invariante la validez del estado, lo cual garantiza que el sistema no puede transicionar a un estado que no represente un posible estado DNSSEC. Sin embargo, también pudo detectarse una inconsistencia en los datos dentro de la cadena de confianza al ejecutarse el rollover de una llave de zona, ya que puede suceder el caso en que los datos almacenados en el caché de un servidor discrepen de los datos verdaderamente publicados.

Agradecimientos

Me gustaría agradecer especialmente a Carlos Luna, sin él esta tesina nunca se hubiese emprendido, y es probable que sin su ayuda, comprensión y motivación este trabajo todavía no habría llegado a su fin.

A mi familia que tanto quiero, a mis queridos abuelos y mi Tío, ellos tienen todo mi afecto y cariño, siempre estuvieron presentes y me acompañaron en todas las etapas de mi vida; también a mis abuelos Carlos y Sara que siempre confiaron y creyeron en mí; a mi tía Vero, Fer, Luci y Santi con los que he compartido tantos momentos alegres. A mis hermanos, Pablo que además es un gran amigo, por bancarme a diario y brindarme siempre su ayuda incondicional, Juan que cada día me alegra y enorgullece más ver la persona en la que se ha convertido, y Danielita que es la hermanita más linda y buena que me pudo haber tocado, y que trae constantemente tanta felicidad a mi vida. Y por supuesto a mis padres Julio y Patricia, que siempre me apoyaron y me dieron todo, porque siempre pusieron a sus hijos antes que a cualquier otra cosa, siempre supieron aconsejarme y motivarme en todos los momentos de mi vida, ellos son un ejemplo y tienen toda mi admiración. Ojalá algún día yo sea la mitad de buen padre que ellos son para mí.

Este es el cierre de una etapa muy importante de mi vida, y seguramente no pueda expresar en estas pocas líneas cuanto significan para mí todas aquellas personas que me acompañaron y ayudaron en el camino, pero mínimamente me gustaría nombrarlos a todos. Quisiera agradecerles a los directivos y profesores por su dedicación y buena voluntad, que junto con su compromiso y buena onda hicieron que mi paso por la facultad sea uno de mis recuerdos más gratos.

No puedo dejar de incluir a todos aquellos que me acompañaron en el aula, en el bar y en tantas horas de estudio... Coco, Fer, Chacho, Tincho, Fabri, Ema, Pablo B. y R., Checho, Bibi, Uciel, Zeta; y por supuesto mis queridísimos Santiago y Taihú. De nuestra querida LCC también tuve el placer de conocer a Cristian y a Luis en este última recta. A Dante, que su ayuda fue muy significativa para poder encaminar este trabajo. Estoy muy agradecido por todos estos compañeros que tuve y que hoy puedo decir sin ninguna duda que los considero grandes amigos.

Mis agradecimientos también son para todos aquellos que me acompañaron fuera del ámbito académico. A mis amigos de siempre, los del San José, también Martín, Diego, Tincho, Sergio, Edi, Chiqui, por dejarme compartir tantos años inolvidables con ustedes.

A mi nueva familia, Silvita y Quique, Nico, Nacho y Marti (entre tantos que son) que siempre me hacen sentir uno más y sé que siempre puedo contar con ellos. Gracias por dejarme ser parte, ustedes se ganaron todo mi cariño.

Finalmente a Lía, por creer en mí y respaldarme siempre. Con ella todo es más fácil y más lindo; y así como tuve su ayuda, cariño y apoyo en este último trecho del camino, así como superamos las distancias, sé que lograremos mucho juntos. Al amor de mi vida, simplemente gracias por estar conmigo y hacerme tan feliz.

Índice general

Capítulo 1

Introducción

El *Sistema de Nombres de Dominio (Domain Name System - DNS)* surgió a partir del crecimiento inesperado de Internet. A finales de la década del 60, la agencia del desarrollo de proyectos del departamento de defensa de los Estados Unidos (Advanced Research Projects Agency - ARPA), comenzó a desarrollar lo que se conoció como ARPAnet[1], una red de computadoras de área amplia experimental, la cuál conectó importantes organizaciones de investigación dentro de los Estados Unidos. El objetivo de la ARPAnet era permitir al gobierno el desarrollo conjunto y la investigación compartida con equipos remotos a partir de escasos y costosos recursos informáticos.

A través de los años 1970's, la *ARPAnet* era una red pequeña, de unos pocos cientos de hosts. Un simple archivo *HOSTS.TXT* contenía el mapeo nombre-a-dirección para cada host conectado a la ARPAnet. Este archivo era mantenido por *SRI's Network Information Center*[2] (apodado NIC) y distribuido desde un único host, SRI-NIC [18] [19]. De esta manera, los administradores de ARPAnet, eventualmente enviaban por correo electrónico sus cambios al NIC, y periódicamente descargaban vía FTP[3] desde SRI-NIC la versión al corriente del archivo HOSTS.TXT, y sus cambios eran compilados a un nuevo archivo HOSTS.TXT una o dos veces a la semana. Con el crecimiento de ARPAnet, este esquema dejó de ser manejable. El tamaño de HOSTS.TXT creció en proporción al crecimiento de hosts en la ARPAnet. Más aún, el tráfico generado por el proceso de actualización se incrementó todavía más rápido, ya que cada host adicional, significaba no sólo otra línea en el archivo HOSTS.TXT, sino también otro potencial host actualizándose de SRI-NIC.

Cuando la ARPAnet comenzó a utilizar el protocolo TCP/IP (Transmission Control Protocol/Internet Protocol), la población de la red explotó, lo cuál significó una serie de problemas con el archivo HOSTS.TXT: *SRI-NIC ya no podía mantener el tráfico de red y la carga de procesador que significaba distribuir el archivo.* No podrían existir dos hosts en el archivo HOSTS.TXT con un mismo nombre. Aunque el NIC podría asignar direcciones de manera de garantizar que sean únicas, no tenía autoridad sobre los nombres de dominio. De esta manera, no había nada que prevenga a alguien agregar un host con

[1] Advanced Research Projects Agency Network
[2] SRI: Stanford Research Institute
[3] File Transfer Protocol

un nombre que cause conflicto con otro, y desmorone todo el esquema.

Se hacía imposible mantener la consistencia del archivo a lo largo de esta red en crecimiento. Para el momento en que un nuevo archivo HOSTS.TXT estaba actualizado en Internet, un host ya podía haber cambiado de dirección o un nuevo host podría haber aparecido. De esta manera, se creó la necesidad de tener un servicio de nombres de propósito general. El resultado fueron varias ideas acerca del espacio de nombres y su gestión [30] [20] [34] [33]. Los propósitos cambiaron, pero la idea común era la de un espacio de nombres jerárquico que se correspondiera con la estructura de la organización y del uso del "." como carácter de unión entre los niveles de la jerarquía. En los [25] [26] se describió un diseño utilizando una base de datos distribuida y recursos generalizados. En base a la experiencia de varias implementaciones, el sistema evolucionó a los [27] [28], dando origen al *DNS (Domain Name System)*. Y con el correr del tiempo, éstos RFC han sido seguidos por muchos otros, que describen potenciales problemas de implementación, seguridad, etc.

En el presente trabajo, nos enfocaremos en ciertos problemas del sistema DNS, por lo que se han diseñado ciertas extensiones de seguridad, dando lugar a *DNSSEC (Domain Name System Security Extensions)* [1] [3] [2].

El objetivo de este trabajo es analizar en profundidad el sistema $DNSSEC$, y obtener una especificación formal del mismo, utilizando el cálculo de construcciones inductivas [22] [12] y el asistente de pruebas COQ [23] [10], posibilitando el planteo de propiedades de invariancia que comprueben que este sistema realmente soluciona aquellos problemas de seguridad a los que se encuentra expuesto su antecesor DNS. Así como también, a partir de la formalización planteada estudiar posibles fallas en el mismo.

Eventuales actualizaciones y material complementario a este trabajo, incluido el desarrollo completo en el asistente de pruebas Coq de la teoría presentada, puede descargarse desde http://www.enexos.com.ar/∼ebazan.

El documento sigue la estructura de [31], organizado en capítulos, donde cada uno desarrolla un aspecto de la formalización utilizando lo definido en los anteriores.

El capítulo 2 presenta conceptos básicos, el formato de un mensaje DNS, y posteriormente sus vulnerabilidades.

El capítulo 3 presenta el modelo DNSSEC.

El capítulo 4 introduce la formalización de DNS propuesta por Cheung-Levitt [32], ampliada para modelar DNSSEC.

En el capítulo 5 se desarrolla la verificación del modelo - invariancia de la validez del estado respecto a la ejecución de los comandos, analizando particularmente la validez de la cadena de confianza definida en el sistema $DNSSEC$.

El capítulo 6 presenta las conclusiones obtenidas a lo largo del desarrollo de la tesina, y plantea posibles trabajos futuros para profundizar en el tema.

Capítulo 2

El Sistema de Nombres de Dominio

El presente capítulo describe el Sistema de Nombres de Dominio *DNS*[1]. Primero se introduce de manera general, presentando conceptos básicos y términos, para utilizar de manera consistente a lo largo del documento. Luego se presenta el formato de un mensaje DNS. Posteriormente se describen las vulnerabilidades del sistema de nombres de dominio.

2.1. Descripción general del modelo DNS

DNS es básicamente una base de datos distribuida e indexada por *nombres de dominio*. Cada nombre de dominio es esencialmente un camino en una estructura de árbol jerárquico, llamado *espacio de nombres de dominio*, como se puede observar en la Figura 2.1.

Cada nodo posee una etiqueta, y cada nombre de dominio no es más que la secuencia de éstas etiquetas, separadas por un ".", desde un nodo en particular hasta la raíz, refle-

[1]Domain Name System

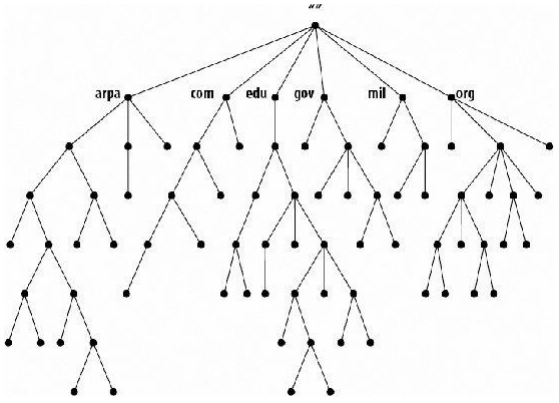

Figura 2.1: Estructura jerárquica del espacio de dominios DNS [21]

jando de ésta manera en los nombres de dominio la jerarquía de la estructura. Para más información sobre cómo funciona DNS, ver [21] [27] [28].

Uno de los objetivos principales de DNS era descentralizar la administración, y esto es logrado gracias a la *delegación*. De manera similar, una organización administrando un dominio, puede dividirlo en subdominios, y cada uno de éstos puede ser delegado a otras organizaciones, haciendo responsable a cada una de éstas organizaciones a mantener todos los datos de su subdominio, siendo libres de cambiar dichos datos e inclusive de dividir su subdominio en más subdominios y delegarlos. El dominio unr.edu.ar por ejemplo, es delegado al administrador correspondiente de la "Universidad Nacional de Rosario" (Figura 2.2).

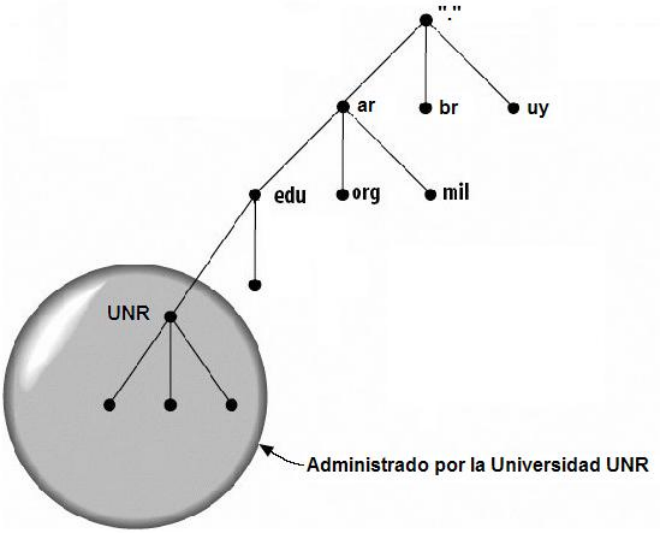

Figura 2.2: unr.edu.ar es delegado a la U.N.R. [21]

Una *zona*, es una parte contigua del espacio de dominio, la cual es administrada por un conjunto de servidores, llamados *nameservers*. Los nameservers que administran una zona tienen *autoridad* sobre la misma.

La diferencia entre dominios y zonas es importante. Muchos dominios, tales como *unr.edu.ar* se subdividen mediante delegación en unidades más pequeñas y manejables. Estas unidades son llamadas zonas (Figura 2.3).

El espectro DNS define dos tipos de servidores nameservers: *servidor primario (o master)* y *servidor secundario (o slave)*. Un servidor primario, lee los datos de una zona directamente de su archivo de hosts. Un secundario, toma los datos autoritativos para la zona desde su servidor primario. Ambos son autoritativos para la zona y a pesar de su nombre, los servidores secundarios no son nameservers de segunda clase, sino que DNS provee estos dos tipos de servidores para una zona dada, por cuestiones de redundancia, o balance de carga.

Los clientes de DNS, aquellos que acceden a los nameservers, son llamados *resolvers*. Un programa corriendo en un host que necesite información del espacio de nombres de

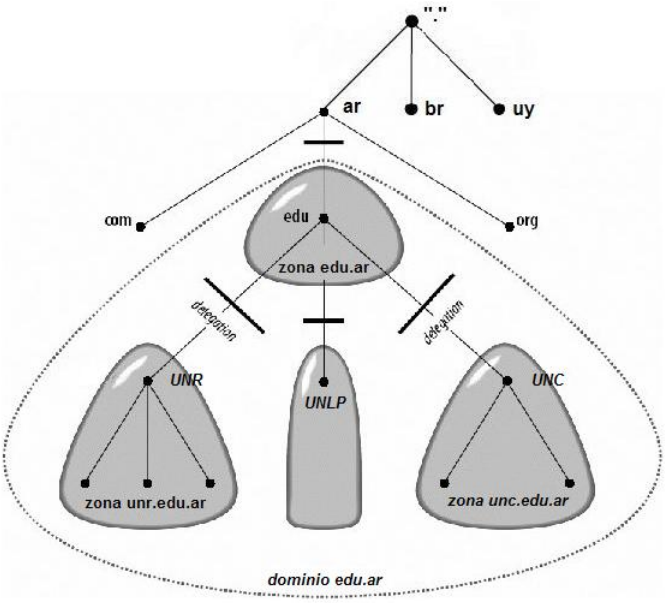

Figura 2.3: El dominio edu.ar subdividido en zonas [21]

dominio utilizará un resolver. De esta manera, un resolver se encargará de:

- Preguntar a un nameserver (Queries)
- Interpretar las respuestas obtenidas (las cuáles pueden ser *Resource Records* o un *error*)
- Devolver la información requerida al programa que la solicitó

El proceso para obtener información de DNS es llamado *resolución de nombres* o simplemente *resolución*. Supongamos que el host *h1.fceia.unr.edu.ar* necesita la dirección IP de *h2.famaf.unc.edu.ar*. Para esto, el resolver deberá solicitar esta información al servidor de nombres local dentro del dominio *fceia.unr.edu.ar*. Existen dos modos de resolución: iterativa y recursiva. En el modo iterativo, cuando un servidor de nombres recibe una petición para la cual no sabe la respuesta, dará referencia al solicitante de otros servidores con más chances de conocer la respuesta. Cada servidor es inicializado con las direcciones de algunos servidores autoritativos para la zona root. Más aún, los servidores root, conocen los servidores autoritativos de los dominios de segundo nivel (e.g., dominio *ar*). Los servidores de segundo nivel conocen los servidores autoritativos para los dominios de tercer nivel, y así sucesivamente. De esta manera, siguiendo la estructura de árbol, el solicitante puede ir *"acercándose"* a la respuesta después de cada referencia obtenida. La figura 2.4 muestra el escenario de resolución iterativo.

Por ejemplo, cuando un servidor root recibe una petición *"iterativa"* para el dominio *h2.famaf.unc.edu.ar*, este referirá al solicitante al servidor *"ar"*, y eventualmente éste localizará al servidor autoritativo para *famaf.unc.edu.ar* y obtendrá su dirección IP.

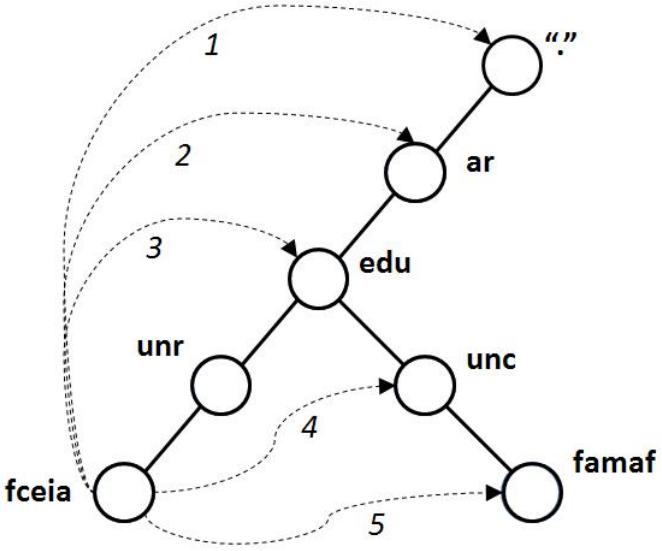

Figura 2.4: Modo iterativo de resolución de nombres [32]

En el modo recursivo, un servidor o bien responde la consulta, o encuentra la respuesta contactando él mismo a otros servidores y devolviendo una respuesta al solicitante.

El proceso completo de resolución puede parecer bastante complicado y engorroso, pero en general, es bastante rápido. Una de las características que acelera considerablemente este proceso es el almacenamiento en caché. Un servidor de nombres procesando una consulta recursiva, puede tener que enviar varias consultas él mismo para poder averiguar la respuesta, y a medida que lo hace descubre mucha información acerca del espacio de nombres de dominio. Cada vez que es referido a otra lista de servidores de dominio, aprende que éstos son autoritativos para una cierta zona, así como también aprende sus direcciones IP. Al terminar el proceso de resolución, cuando finalmente encuentra la información que le han solicitado, puede almacenar esta información para futuras consultas. De esta manera, la próxima vez que un resolver consulte al servidor de nombres por información que ya ha sido solicitada previamente, dicho servidor tendrá guardada la respuesta en caché y simplemente brindará la respuesta correspondiente.

Los servidores de dominio no pueden guardar información en caché por siempre, ya que si lo hicieran, los cambios en los datos de los servidores autoritativos nunca se propagarían, dado que los servidores de nombres remotos seguirían utilizando sus datos en caché. Por lo tanto, el administrador de la zona que contiene los datos autoritativos decide un *time to live (ttl)* para esta información. Dicho *ttl* es la cantidad de tiempo que cualquier servidor de nombres puede guardar la información en caché. Luego de expirado éste valor, el servidor de nombres debe descartar la información guardada en caché y requerir nueva información de los servidores autoritativos.

2.2. Formato del mensaje DNS

Un mensaje DNS consiste de un encabezado y cuatro secciones: *question, answer, authority y additional*. (Figura 2.5)

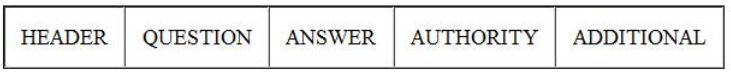

Figura 2.5: Formato de un mensaje DNS [13]

La primer sección *Header*, contiene un campo id, el cuál es usado para reconocer una respuesta coincidente con una consulta previamente realizada.

La sección *question* consta de un nombre de dominio (*QNAME*), un tipo (*TYPE*), y una clase (*QCLASS*). Por ejemplo, una consulta para averiguar la dirección IP del host *h1.fceia.unr.edu.ar* posee *QNAME=h1.fceia.unr.edu.ar*, *QTYPE=A*, y *QCLASS=IN* (IN denota Internet).

Las últimas tres secciones mencionadas (*answer, authority y additional*) están compuestas por Resource Records (RR's). A su vez, un RR DNS tiene 6 campos: *NAME, TYPE, CLASS, TTL, RDLENGTH, RDATA*.

El campo *NAME* contiene su nombre DNS, el cual también hace referencia al dueño al cual pertenece este RR.

El campo *TYPE* indica el tipo del RR, la siguiente es una lista de los tipos más comunes:

- **A**: contiene una dirección IP de 32 bit para el nombre de dominio especificado.
- **CNAME**: especifica el nombre de dominio original (o canónico) para el nombre de dominio especificado (Mapea un alias con su nombre de dominio canónico).
- **HINFO**: contiene información del host, tal como ser el sistema operativo usado.
- **MX**: contiene un nombre de host actuando como servidor de correo para el nombre de dominio especificado.
- **NS**: contiene un nombre de host actuando como servidor de nombres de dominio autoritativo para el nombre de dominio especificado.
- **PTR**: contiene un nombre de dominio correspondiente a una dirección IP especificada.
- **SOA**: contiene información sobre el dominio especificado (por ejemplo, el correo electrónico del administrador del dominio).

El campo *CLASS* IN es utilizado para Internet. Si bien existen otras clases son omitidas por brevedad.

El campo *TTL*, especifica el tiempo (en segundos) que un servidor de nombres puede almacenar un RR en caché.

El campo *RDATA* contiene los datos propiamente dichos del RR, y está definido específicamente para cada tipo de RR.

Estos RR's están agrupados en conjuntos (*RRSets*). Un RRSet contiene 0 o más RRs [36], que tienen los mismos valores en los campos NAME, CLASS y TYPE, pero los datos RDATA son distintos. La figura 2.6 muestra un ejemplo de RRSet.

```
example.com.  IN  NS  ns1.example.com.
example.com.  IN  NS  ns2.example.com.
example.com.  IN  NS  ns.plain.org.
```

Figura 2.6: Estos tres RRs están agrupados en un mismo RRSet [13]

2.3. Vulnerabilidades en el sistema DNS

La especificación original de DNS no incluía seguridad, ya que fue diseñado para ser una base de datos pública en la cual el concepto de restringir el acceso a la información dentro del espacio de nombres no era parte del protocolo. Con el crecimiento de aplicaciones desarrolladas en base al uso de direcciones IP y hostnames para permitir o denegar acceso, se comenzó a demandar precisión en la información contenida en DNS. Información falsa dentro de DNS puede llevar a problemas inesperados y potencialmente peligrosos. Estas vulnerabilidades pueden etiquetarse dentro de las siguientes categorías: *envenenamiento de caché, inundación de clientes, vulnerabilidades en actualizaciones dinámicas, fuga de información, y base de datos comprometida en un servidor DNS* [13]. (Figura 2.7).

Varios autores han discutido acerca de los problemas de seguridad de DNS [8] [9] [11] [17] [29] [35]. Siguiendo la línea de trabajo de Cheung-Levitt [32], nos centraremos en dos problemas específicos de DNS relevantes para el desarrollo de DNSSEC: ***envenenamiento de caché (cache poisoning) y falta de autentificación de respuestas DNS***.

2.3.1. Cache poisoning

Cuando un servidor DNS no tiene dentro de su caché la respuesta a una consulta, puede re-preguntar dicha consulta a otro servidor DNS en beneficio del cliente. Si esto sucede el servidor consultado posee información incorrecta, ya sea intencionalmente o por error, se dice que ocurre envenenamiento de caché. Cuando el envenenamiento de caché es intencional se refiere a éste como *DNS spoofing*.

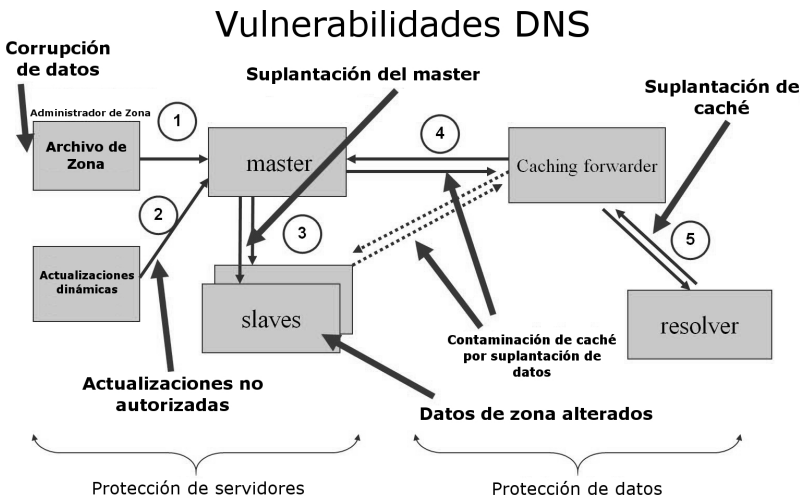

Figura 2.7: Vulnerabilidades DNS [24]

Cuando el ataque es malicioso, el atacante puede engañar a un servidor de nombres **S1** para preguntar a otro servidor **S2**. Si **S2** es un servidor comprometido, el atacante puede lograr tanto *denegación de servicio (denial of service - DoS)* y brindar respuestas negativas, así como enmascararse como una entidad confiable y dar respuestas que contengan RRs falsos. De cualquier manera, como los servidores almacenan los resultados de las interacciones previas con otros servidores para mejorar su performance, cuando **S1** use su caché contaminado para resolver un nombre de dominio, podrá estar usando información incorrecta suministrada por el atacante. (Figura 2.8).

2.3.2. Falta de autentificación de respuestas DNS

El mecanismo de autenticación de mensajes en DNS es muy débil: un servidor DNS (o cliente DNS) adjunta un id al realizar una consulta, el cuál se utilizará posteriormente para hacerlo coincidir al recibir la respuesta correspondiente. Por ejemplo, supongamos que un servidor **S1** envía una consulta a otro servidor **S2**. Si un atacante puede predecir el id utilizado por **S1** al realizar dicha consulta, éste podrá enviar una respuesta falsificada con el id coincidente. Cuando **S1** reciba la respuesta, supuestamente proveniente de **S2**, **S1** no tiene manera de verificar que la respuesta efectivamente provenga de **S2**. Si **S2** no está disponible cuando la consulta es enviada, el atacante puede tomar su lugar y enviar esta respuesta falsificada, e inclusive si **S2** está operacional, el atacante puede montar un ataque de denegación de servicio contra **S2** para prevenir a **S2** de responder a la consulta de **S1**. El atacante también puede valerse del hecho de que si un servidor recibe múltiples respuestas para su consulta, utilizará la primera en recibir. De esta manera, inclusive si **S2** puede responder a **S1**, el atacante puede tener éxito si su respuesta falsificada llega a **S1** antes que la de **S2** lo haga. (Figura 2.9).

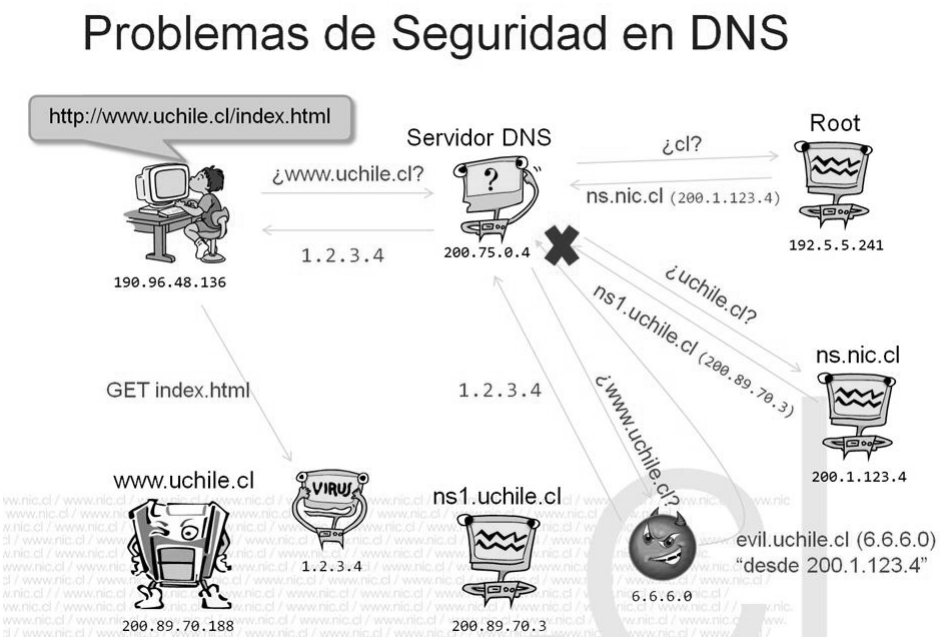

Figura 2.8: Envenenamiento de caché [4]

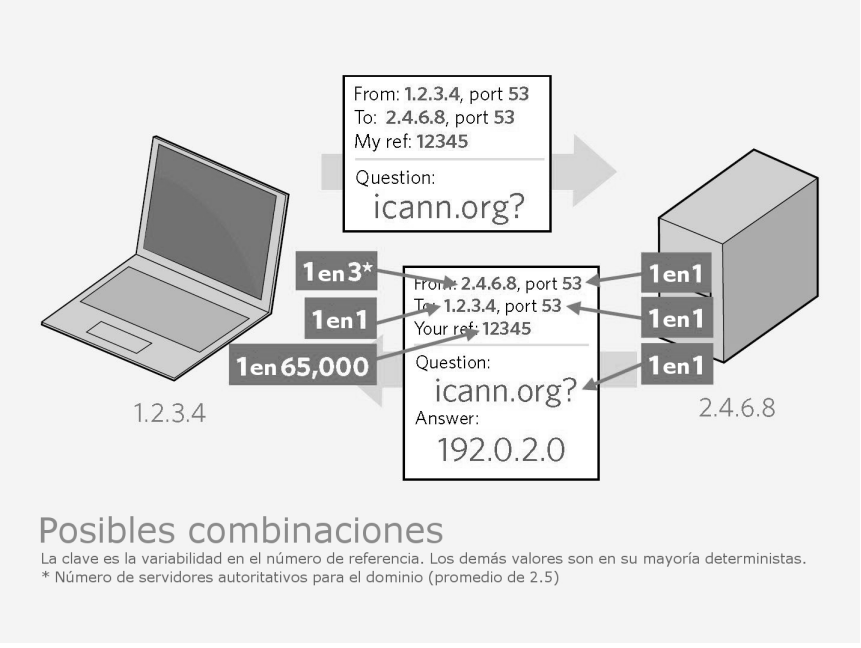

Figura 2.9: Suplantación de identidad [14]

Capítulo 3

El modelo DNSSEC

En 1994, la IETF[1] formó un grupo de trabajo para proveer extensiones de seguridad al protocolo DNS en respuesta a los problemas de vulnerabilidad mencionados en la sección anterior. Estas extensiones de seguridad, *DNSSEC*[2], fueron diseñadas para co-existir, operando correctamente con implementaciones no seguras de DNS. Para lograr esto, se diseñaron nuevos tipos de RRs[3], los cuáles fueron usados en conjunto con los ya existentes, permitiendo una menor complejidad de migración a servidores en producción; así como también una buena adaptación de funcionamiento entre clientes seguros y no seguros, facilitando el correcto procesamiento de RRSets[4] recibidos de un servidor, ya sea DNS o DNSSEC [16] [7].

En el presente capítulo se introduce el modelo DNSSEC y su alcance, las diferencias en el formato del mensaje con respecto a DNS y los nuevos registros (RRs) introducidos, luego se describe la seguridad llevada a cabo en los servidores, así como también la cadena de confianza que se genera en todo el árbol DNS utilizando DNSSEC. Finalmente, se presenta un ejemplo, explicando una consulta típica de transacción DNSSEC.

3.1. Objetivos de DNSSEC

Un principio fundamental de DNS es que si bien se requiere información correcta y consistente, dicha información debe ser pública. De esta manera, sí existe la necesidad de autenticación e integridad, pero no para realizar control de acceso o por confidencialidad. Los objetivos de **DNSSEC** son justamente, proveer autenticación e integridad al sistema DNS, lo cual se llevó a cabo a través de firmas criptográficas generadas mediante el uso de llaves públicas-privadas. Por esto, servidores, resolvers y aplicaciones con manejo de seguridad pueden hacer uso de estas ventajas, y asegurar que la información obtenida de un servidor **DNS seguro** es auténtica y no ha sido alterada. Incluso aplicaciones

[1]Internet Engineering Task Force
[2]Domain Name System Security Extensions
[3]Resource Records
[4]Resource Records Sets

fuera de DNS pueden decidir utilizar las llaves públicas contenidas dentro del sistema para proveer confidencialidad. Cabe destacar que DNSSEC brinda los mecanismos para manejar múltiples llaves, cada una generada con un algoritmo de encriptación distinto para un nombre DNS dado.

3.2. Alcance de DNSSEC

El alcance de las extensiones de seguridad a DNS pueden resumirse en tres servicios: *distribución de llaves*, *autenticación del origen de los datos*, y *autenticación de transacciones y pedidos*.

- **Distribución de llaves (key distribution)**: El servicio de distribución de llaves no sólo permite la recuperación de una llave pública de un nombre DNS para verificar la autenticidad de los datos de una zona DNS, sino que también proporciona un mecanismo mediante el cual cualquier llave asociada a un nombre DNS puede ser usada para otros propósitos. El servicio de distribución de llave pública soporta diferentes tipos de llaves y de algoritmos de encriptación.

- **Autenticación del origen de los datos (Data Origin Authentication)**: Este servicio es el punto crucial en el diseño de DNSSEC. Mitiga amenazas tales como *envenenamiento de caché y compromiso de los datos de una zona en un servidor DNS*. Los RRSets dentro de una zona son firmados criptográficamente, dando así un alto nivel de certeza de que los datos recibidos por resolvers y servidores pueden ser confiables. DNSSEC utiliza firmas digitales para firmar los RRSets dentro de DNS. Dicha firma digital contiene un valor hash[5] cifrado del RRSet, el cual es una suma de control criptográfica de los datos contenidos en el mismo. Este hash es firmado (digitalmente encriptado) mediante una clave privada, generalmente perteneciente al dueño de la información, conocido como *signer (o signing authority)*. El receptor del RRSet puede simplemente chequear la firma digital contra los datos en el RRSet, primero desencriptando la firma digital (RRSIG[6]) utilizando la llave pública del signer para obtener el hash original de los datos, y luego computando su propio hash de los datos en el RRSet utilizando el mismo algoritmo de suma de control criptográfica De esta manera puede comparar los resultados del hash encontrado en la firma digital contra el hash recién computado. Si los dos valores hash coinciden, existe integridad en los datos y el origen de datos es auténtico.

- **Autenticación de transacciones y pedidos (DNS Transaction and Request Authentication)**: Permite autenticar pedidos y cabeceras de mensajes DNS. Esto garantiza que una respuesta sea verdaderamente en respuesta a una consulta realizada, y que dicha respuesta proviene efectivamente desde el servidor al cual estaba destinada la consulta. Proporcionar la seguridad para ambos se hace en un solo paso, parte de la información regresada desde un servidor seguro es una firma, dicha

[5]Un valor hash se utiliza para resumir o identificar probabilísticamente un gran conjunto de información
[6]nuevo tipo de RR introducido en DNSSEC, se discutirá en detalle durante el capítulo

firma es producida por la concatenación de la consulta y su respectiva respuesta. Esto permite a un resolver seguro, realizar cualquier verificación concerniente a la transacción realizada.

Esta autenticación también se utiliza en las actualizaciones dinámicas de DNS. Sin DNSSEC, las actualizaciones dinámicas no proveen ningún mecanismo que prohíba, a cualquier sistema que tenga acceso a un servidor autoritativo DNS, la actualización de información de su zona. Para proveer seguridad para estas modificaciones se incorpora DNSSEC para brindar una autentificación fuerte a sistemas habilitados a manipular dinámicamente información de zonas DNS en el servidor primario [15].

3.3. Modificaciones en el encabezado DNS

Se modifica el encabezado de un paquete DNS, de manera de agregar el bit **DO**[7], y los bits **CD**[8] y **AD**[9].

- **DNSSEC OK (DO)**: Habilitando este bit en una consulta se le indica al servidor que el resolver está habilitado a aceptar RRs DNSSEC. El bit DO deshabilitado indica que el resolver no soporta las extensiones de seguridad de DNSSEC y sus correspondientes RRs *NO DEBEN* ser retornados en la respuesta. El bit DO *DEBE* ser copiado en la respuesta.

  Un servidor recursivo que soporte DNSSEC, habilita el bit DO en los pedidos recursivos, sin importar el estado de dicho bit en el pedido inicial del resolver. Si el pedido inicial hecho por el resolver no tiene el bit DO habilitado, el servidor que SI soporta DNSSEC *DEBE remover los RRs de seguridad* de DNSSEC antes de devolver los datos al cliente, de todos modos los datos en caché *NO DEBEN* ser modificados.

- **Checking Disabled (CD)**: Este bit es controlado por los resolvers, e indica que el mismo realizará las autenticaciones dadas por sus políticas de seguridad local. Al habilitarlo, el servidor de nombres deberá responder los RRs solicitados, inclusive si no pasan sus políticas de seguridad.

- **Authenticated Data (AD)**: Este bit es habilitado por un servidor seguro, sólo si todos los datos que se incluyen al dar una respuesta han sido criptográficamente verificados o cumple las políticas de seguridad del servidor local.

[7]DNSSEC OK
[8]Checking Disabled
[9]Authenticated Data

3.4. Nuevos Registros

DNSSEC agrega los siguientes registros (Resource Records) al protocolo DNS [1] [3] [2]:

- **DNSKEY**
- **RRSIG**
- **NSEC**
- **DS**

3.4.1. DNSKEY

DNSSEC utiliza criptografía de llave pública para firmar y autenticar RRsets. Las llaves públicas son guardadas en el RR DNSKEY y son utilizadas en el proceso de autenticación de DNSSEC: Una zona firma sus RRsets utilizando una llave privada y guarda la correspondiente llave pública en el RR DNSKEY. De esta manera, un resolver puede utilizar esta llave pública para validar las firmas que cubren los distintos RRsets en la zona (RRSIG), y de esta manera autenticarlas. En la figura 3.1 se puede observar el formato de los datos (RDATA) del RR DNSKEY.

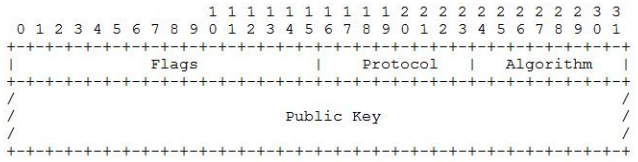

Figura 3.1: Formato RDATA para DNSKEY [3]

3.4.2. RRSIG

Las firmas digitales son guardadas en los RRs RRSIG y son usadas en el proceso de autenticación de DNSSEC. Para realizar una validación se puede usar estos RRs para autenticar RRsets de la zona en cuestión. Los RRs RRSIG DEBEN tener material de verificación (firmas digitales) usadas sólo para asegurar las operaciones de DNS.

Un RR RRSIG contiene las firmas para un RRset con un nombre, clase y tipo particular. Los RR RRSIG especifican un intervalo de validez para la firma y usa los campos *Algorithm, Signer's Name y Key Tag* para identificar el registro de la llave pública DNSKEY que se pueden usar para verificar la firma.

Como todo RRset autoritativo en una zona debe ser protegido por una firma digital, los RR RRSIG también deben estar presentes para nombres que contienen un RR CNAME,

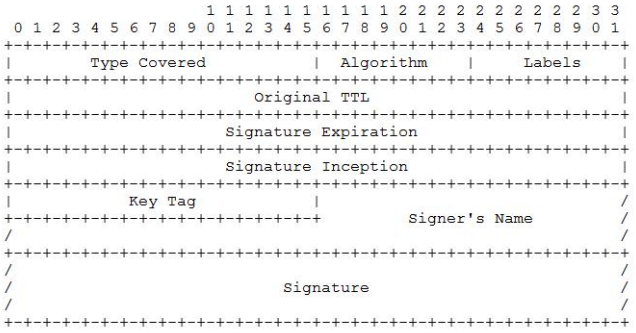

Figura 3.2: Formato RDATA para RRSIG [3]

éste es un cambio a la especificación de DNS tradicional, la cual establece que si un CNAME está presente para un nombre, es el único tipo permitido en ese nombre. Sin embargo, en una zona firmada los RR RRSIG y NSEC DEBEN existir para el mismo nombre que el RR CNAME.

En la figura 3.2 se puede observar el formato de los datos (RDATA) del RR RRSIG.

3.4.3. NSEC

El RR NSEC nos indica dos cosas: el próximo nombre (en el orden canónico de la zona) que contenga tanto datos autoritativos como un punto de delegación NS RRset, y el conjunto de tipos de RRs presentes para el dueño del record NSEC. El conjunto completo de RRs NSEC en una zona indican cuáles RRSets autoritativos existen en una zona, así como también forman una cadena de dueños de nombres autoritativos en la zona. Esta información es usada para proveer denegación de existencia autenticada para los datos DNS.

Como todo RRSet autoritativo en una zona, debe ser protegido por una firma digital, los registros RRSIG deben estar presentes para nombres que contienen un registro CNAME, éste es un cambio a la especificación de DNS tradicional, la cual establece que si un CNAME está presente para un nombre, es el único tipo permitido en ese nombre, en una zona firmada los registros RRSIG y NSEC DEBEN existir para el mismo nombre que el RR CNAME.

En la figura 3.3 se puede observar el formato de los datos (RDATA) del RR NSEC.

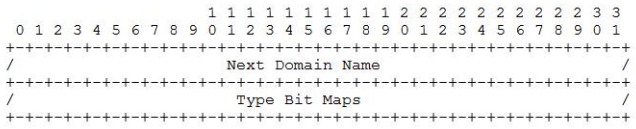

Figura 3.3: Formato RDATA para NSEC [3]

3.4.4. DS

El RR DS hace referencia a un RR DNSKEY y es usado en el proceso de autenticación de DNSSEC. Un RR DS hace referencia a un RR DNSKEY guardando su respectiva *"etiqueta de la clave"* (key tag), *"número de algoritmo"* y un *resumen del RR DNSKEY*. Notar que si bien el resumen debe ser suficiente para identificar la llave pública, guardar el key tag y el algoritmo ayuda a hacer el proceso de identificación más eficiente. Autenticando el registro DS, un resolver puede autenticar el RR DNSKEY al cual el RR DS está apuntando.

El RR DS y su correspondiente RR DNSKEY tienen el mismo nombre de dueño, pero son guardados en diferentes servidores. El RR DS aparece sólo en la parte superior (padre) de la delegación y es dato autoritativo en la zona del padre. Por ejemplo, el RR DS para *"example.com"* es guardado en la zona *"com"*. El correspondiente RR DNSKEY es guardado en la zona *"example.com"*. Esto simplifica la administración de la zona DNS y la firma de la zona pero introduce requerimientos de procesamiento de respuestas especiales para el RR DS.

En la figura 3.4 se puede observar el formato de los datos (RDATA) del RR DS.

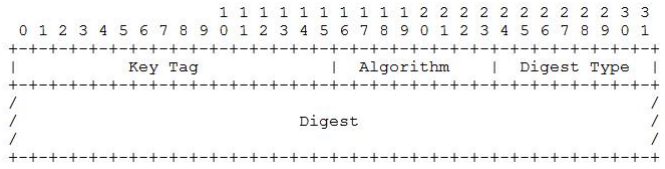

Figura 3.4: Formato RDATA para DS [3]

3.5. Seguridad en los servidores DNS

Todo servidor DNS primario tiene 3 funciones principales: ***administrar información autoritativa de su zona, administrar el almacenamiento en caché de la información DNS, y responder consultas***. En DNSSEC, todo servidor seguro tiene responsabilidades añadidas a cada una de estas funciones [13].

En la administración de los datos de la zona, se deben incluir los nuevos RRs RRSIG, NSEC, DNSKEY y DS. Los RR RRSIG son generados para los RRSets de la zona. La llave privada de la zona utilizada para generar estas firmas es mantenida off-line, de manera que su integridad esté menos comprometida y la validez de los datos de la zona sea más confiable. Dicha llave privada es cambiada periódicamente para re-firmar todos los RRs encontrados dentro de la zona. Una vez que los nuevos RRSIG son generados, estos son incluidos dentro del archivo master de la zona. Los RRs NSEC también deben ser generados y ubicados dentro del archivo master de la zona al generar los RRs RRSIG.

Un servidor seguro debe administrar el correcto almacenamiento en caché de la infor-

mación. La responsabilidad añadida para esta tarea comienza en mantener cuatro estados de caché. Un estado, el cual tiene un estado correspondiente en un servidor no-seguro, es *"Bad" (malo)*. En un servidor no-seguro cuando una respuesta *"mala"* es recibida, en la cual la información contenida se encuentra de alguna manera corrupta, dicho servidor no-seguro descarta este mensaje de respuesta sin almacenarlo en caché (y típicamente loguea el evento). En un sentido análogo, un servidor seguro puede descartar cualquier respuesta mala, pero en este caso, una respuesta mala significa que las verificaciones del RR RRSIG con respecto a los datos han fallado. Aunque el RRSet en la respuesta parezca legítimo, este error de chequeo de datos con respecto a su firma es una condición fatal.

Los otros tres estados son: *"Insecure"(Inseguro), "Authenticated" (Autenticado) y "Pending" (Pendiente)*. *Insecure* significa que no hay datos disponibles para verificar la autenticidad de los datos (por ejemplo que estos datos hayan sido obtenidos a partir de un servidor no-seguro). *Authenticated* significa que los RRSets han sido correctamente validados a través del uso de su firma RRSIG y su llave DNSKEY. *Pending* significa que los datos en caché todavía se encuentran en el proceso de verificación de autenticidad.

Otra responsabilidad del servidor es *cuándo* descartar un RRSet del caché. Una vez un RRSet se encuentra almacenado en caché, comienza una cuenta regresiva desde su TTL original, y una vez que llega a cero, éste RRSet es removido del caché. Para servidores seguros, esto ha cambiado un poco. En éstos, no se puede depender sólo del parámetro TTL para descartar un RRSet del caché, sino que se tienen en cuenta dos nuevos tiempos para determinar esta expiración de datos. Los nuevos tiempos son usados para determinar el período de validez de la firma de un RRSet autenticado, y no sólo la expiración del mismo. Estos nuevos tiempos son almacenados dentro del RR RRSIG y son conocidos como *"Signature Inception Time" (tiempo de inicio de la firma)* y *"Signature Expiration Time" (tiempo de expiración de la firma)*. Para servidores y clientes con uso de seguridad, esta información es mucho más relevante para basar la expiración de un RRSet, ya que en este lapso será que éste sea criptográficamente correcto. Aunque el tiempo de expiración de la firma sea correlativo con el TTL, por problemas de compatibilidad el campo TTL no puede ser eliminado. Además, el "envejecimiento" del TTL todavía es incorporado para la eliminación de un RRSet del caché, ya que si un TTL expira antes que el tiempo de expiración de su firma, el RRSet es descartado de todas maneras. Si el tiempo de expiración de la firma ocurre antes de que expire el TTL, entonces dicho TTL será reajustado al tiempo de expiración de la firma y se procederá con la cuenta regresiva de forma normal.

Al utilizar DNSSEC, las respuestas pueden estar dirigidas tanto a clientes inseguros, como a clientes con uso de seguridad. Cuando un resolver inseguro envía una consulta a un servidor seguro, pidiendo información perteneciente a una zona segura, éstos servidores pueden responder sólo con información en caché de tipo Authenticated o Insecure. Un servidor seguro puede enviar datos dentro de caché pertenecientes al estado Pending, sólo cuando la bandera checking disabled (CD) se encuentra habilitada. Un resolver no-seguro, que no participe de DNSSEC, nunca habilitará su bandera CD en una consulta. Ya que enviar datos del tipo Insecure, es lo mismo que utilizar DNS sin DNSSEC, un resolver no-seguro podrá procesar esta respuesta como un mensaje normal de DNS. Al recibir datos del tipo Authenticated, un resolver no-seguro básicamente ignorará la información adicional

referente a seguridad y continuará procesando el mensaje de respuesta del modo usual. Por otro lado, cuando las consultas son originadas desde un resolver seguro, es recomendado habilitar la bandera CD. Con esta bandera habilitada los servidores seguros pueden enviar datos de tipo Pending, logrando dos cosas: minimizar el tiempo de respuesta para manejar consultas, y permitir al resolver implementar sus propias políticas sobre cómo manejar datos de tipo Pending independientemente del servidor. Si los datos de respuesta a la consulta ya son de tipo Authenticated, el servidor habilita la bandera authenticated data (AD) para indicar al resolver que los chequeos necesarios ya han sido realizados, y de esta manera el resolver no necesita hacer ningún tipo de chequeo para verificar la seguridad.

3.6. Cadena de confianza

Los datos de una zona serán confiables si:

- Se encuentran firmados por la llave de la zona *zone-signing-key (zsk)*, mediante los respectivos RRs RRSIG.
- La zone-signing-key de la zona será confiable si está firmada por la llave *key-signing-key (ksk)*, ambos RRs DNSKEY.
- La key-signing-key será confiable si está siendo apuntado por su respectivo RR DS del padre.
- El RR DS será confiable si:
  - está firmado por la zone-signing-key del padre, o
  - los records DS o DNSKEY fueron intercambiados fuera de banda y almacenados localmente (*"Secure entry point"*).

3.7. Ejemplo de cadena de confianza

Para entender mejor como se arma la cadena de confianza del sistema $DNSSEC$, y como podemos recorrerla a la hora de realizar una búsqueda iterativa, utilizaremos el siguiente ejemplo, citado de [24]. Como puede observarse en la figura 3.5, comenzaremos recorriendo la cadena de confianza por un punto conocido por todos: el servidor Root ".", y veremos como se arma la seguridad hasta llegar a "www.ripe.net".

La llave ksk del servidor *Root* **"."** se encuentra manualmente configurada en todos los servidores como una *"llave confiable"*. Esta llave ksk (8907) es utilizada para firmar la llave de zona zsk (2983). El record DS del hijo ***"net"*** es firmado mediante esta llave zsk. Dicho registro DS en la zona de "root" apunta a la llave ksk 7834) de la zona "net". Con la llave ksk, la llave zsk (5612) de la zona "net" es firmada. Esta llave zsk a su vez es utilizada para firmar el registro DS de ***"ripe.net"***.

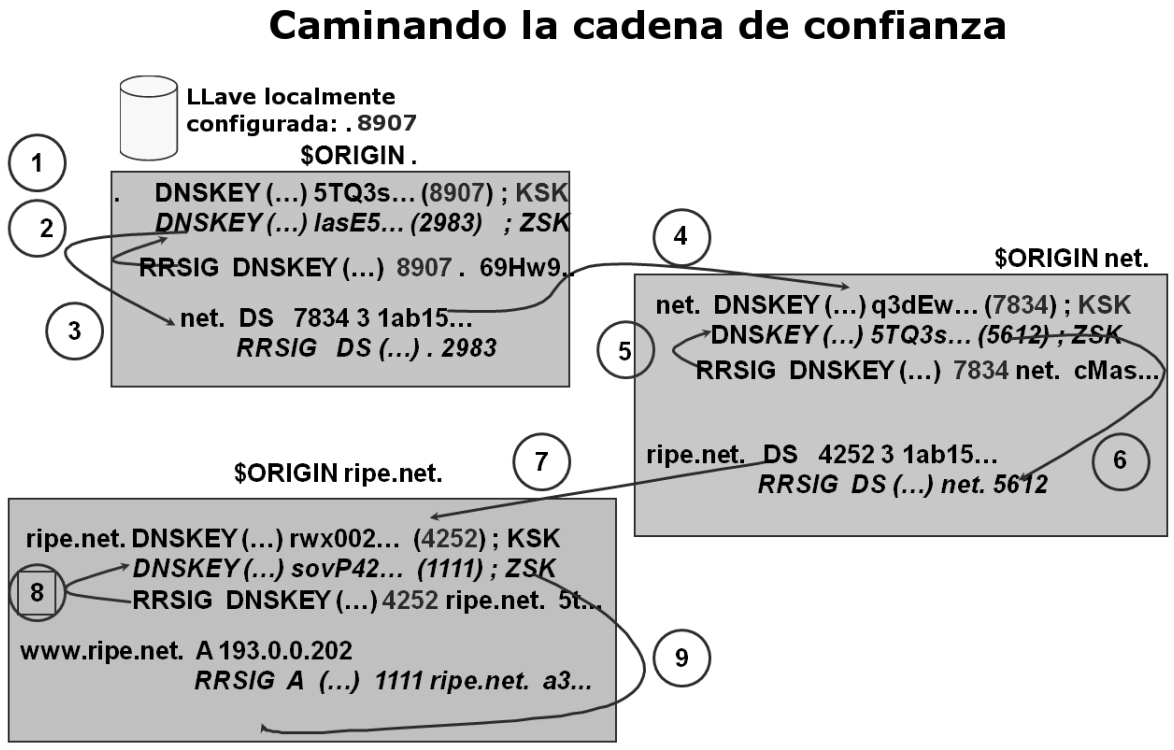

Figura 3.5: Recorriendo la cadena de confianza [24]

Nuevamente, el registro *DS* en el padre "net" es utilizado para apuntar a la llave *ksk* (4252) de "ripe.net". Esta llave *ksk* firma la correspondiente llave de zona *zsk* (1111). Finalmente, la llave *zsk* se utiliza para firmar todos los datos de la zona, incluyendo el registro ***A*** para ***"www.ripe.net"***.

Capítulo 4

Especificación Formal

En el presente capítulo se desarrolla la formalización en el CCI del modelo DNSSEC descripto en el capítulo 3.

En primer lugar se introduce la notación formal utilizada a lo largo de la especificación, luego se presenta la formalización de los componentes básicos, organizada en secciones que comprenden la composición del estado y sus propiedades de validez, posteriormente se describe la sintaxis y semántica de los comandos, y finalmente el concepto de ejecución de estos en el sistema.

4.1. Notación

A lo largo de la presente tesina se utiliza la notación propuesta en [6] que se describe a continuación. Para los conectivos lógicos y de igualdad se utiliza la notación estándar ($\wedge, \vee, \rightarrow, \neg, \exists, \forall$).

Los predicados anónimos son introducidos mediante notación lambda, por ejemplo $(\lambda x.x = 0)$ es un predicado que aplicado a n devuelve true si y solo si n es cero.

Los tipos records son introducidos de la siguiente manera:

$$R \stackrel{\text{def}}{=} [\![\ field_0 : A_0, \ldots, field_n : A_n\ ]\!]$$

lo cual genera un tipo inductivo no recursivo con un único constructor, llamado $Build_R$ y n funciones de proyección $field_i : R \rightarrow A_i$.

Se utiliza $\langle a_0, \ldots, a_n \rangle$ en lugar de $Build_R\ a_0 \ldots a_n$ cuando el tipo R es evidente del contexto. Las aplicaciones de las funciones de proyección están abreviadas usando la notación punto (ej. $field_i\ r = r.field_i$).

La notación para los conjuntos de tipo A es $set\ A$, donde set es un tipo inductivo que codifica los conjuntos como listas cuyos constructores son $cons : A \rightarrow set\ A \rightarrow set\ A$ y $nil : set\ A$, aunque puede suceder que se utilice :: en lugar de $cons$ y [] en lugar de nil. Se utiliza $\{a_0, \ldots, a_n\}_A$ para denotar el conjunto de elementos $a_0, \ldots, a_n$ de tipo A. Para los operadores de conjuntos se utiliza la notación clásica $(\cup, \cap, \in, \subseteq)$.

Para los constructores de tipos inductivos de la forma

$$I : t_0 \ldots t_n \stackrel{\text{def}}{=} C_i\ (a_0 : k_0) \ldots (a_j : k_j) : I\ (b_0 : t_0) \ldots (b_n : t_n) \qquad \forall\ i, j, n \in \mathbb{N}_0$$

puede utilizarse la notación

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

donde las variables libres se consideran universalmente cuantificadas y sus tipos definidos por el contexto.

4.2. Formalización de componentes

4.2.1. Preludio

Para la formalización del sistema se hará uso del desarrollo hecho por [32] para DNS, adaptándolo y ampliándolo para poder describir aspectos claves del sistema DNSSEC.

Primero se introducen algunos tipos utilizados en el resto de la especificación.

En primer lugar, se llaman $DName$, $RRType$, $RRClass$, TTL, $RDLength$ y $RData$ a los tipos Name, Type, Class, TTL, RDLENGTH y RDATA respectivamente, que son los campos básicos de un "Resource Record", el cual se describe como el siguiente Record:

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

A su vez, dentro del tipo de datos $RRType$, podemos notar los nuevos tipos de registro definidos para DNSSEC:

$$\begin{aligned} \mathbf{RRType} \stackrel{\text{def}}{=}\ & A \\ |\ & PTR \\ |\ & NS \\ |\ & CNAME \\ |\ & MX \\ |\ & SOA \\ |\ & HINFO \\ |\ & NSEC \\ |\ & DS \\ |\ & DNSKEY \\ |\ & RRSIG \end{aligned}$$

Para el conjunto de RRs, llamados RRset, definimos la variable **RRset : set RR**.

Para nuestra modelización del sistema DNSSEC, definiremos "RRs Seguros" (SecRR), utilizando el siguiente tipo:

$$\mathbf{SecRR} \stackrel{\text{def}}{=} [\![\ RR' : RR,\ Rsign : RR\ ]\!]$$

donde **Rsign** contiene la firma del correspondiente **RR'**.

A su vez, definiremos un "conjunto de RRs seguros" (SecRRset), de la siguiente manera:

$$\mathbf{SecRRset} \stackrel{\text{def}}{=} [\![\ RRset' : RRset,\ RRsign : RR\ ]\!]$$

donde **RRsign** contiene la firma del correspondiente conjunto **RRset'**.

DNS maneja una base de datos distribuida. Esta base de datos se encuentra indexada por la tupla **(dname, type, class)**, la cual estará definida según el siguiente record, al que llamaremos Idx:

$$\begin{aligned}\mathbf{Idx} \stackrel{\text{def}}{=} [\![\ & Idname : DName,\\ & Itype : RRType,\\ & IClass : RRClass\ ]\!]\end{aligned}$$

El rango de esta base de datos es un conjunto de RRs. Para denotar el tipo de esta base de datos utilizaremos la siguiente función:

$$\mathbf{DbMap} \stackrel{\text{def}}{=} Idx \to SecRRset$$

Los servidores DNSSEC distribuidos globalmente son representados mediante el tipo **Process**.

4.2.2. Mensaje DNS

El sistema DNSSEC, al igual que DNS, realiza su comunicación para el proceso de resolución de nombres a través de mensajes con un formato específico [28].

Un mensaje DNS consiste de un encabezado (Header) y cuatro secciones: Consulta (Question), Autoritativa (Authority) y Adicionales (Additional). Se define utilizando el siguiente Record:

$$\begin{aligned}\mathbf{DNSMessage} \stackrel{\text{def}}{=} [\![\ & Hdr : Header,\\ & Q : Idx,\\ & Ans : SecRRset,\\ & Auth : SecRRset,\\ & Additional : SecRRset\ ]\!]\end{aligned}$$

4.2.3. Envíos pendientes

Para modelar la necesidad de generar envíos pendientes dentro del sistema, ya sean de solicitudes o respuestas, definimos el siguiente Record:

$$\textbf{SendQR} \stackrel{\text{def}}{=} [\![\ SendQ : set\ InfoMsg,\ SendR : set\ InfoMsg\ ]\!]$$

donde **SendQ** y **SendR** hacen referencia al envío de solicitudes (*Queries*) y respuestas (*Responses*) pendientes. El tipo de datos $InfoMsg$ que describe la información necesaria para poder luego ejecutar estos envíos pendientes está definido con el siguiente Record:

$$\textbf{InfoMsg} \stackrel{\text{def}}{=} [\![\ MFrom : Process,\ MTo : Process,\ MMsg : DNSMessage\ ]\!]$$

4.2.4. Llaves del sistema

El sistema DNSSEC basa su seguridad en la utilización de llaves públicas y privadas. Como explicamos anteriormente, cada servidor tiene dos conjuntos de llaves, llaves ZSK y KSK. Si bien estas llaves no son más que RRsets dentro de la base de datos de DNSSEC, para simplificar su uso, las llaves (DNSKEY) del sistema están definidas mediante el siguiente record:

$$\textbf{keySet} \stackrel{\text{def}}{=} [\![\ key : RR,\ ksign : RR\ ]\!]$$

donde key no es más que un RR de tipo DNSKEY, y ksign un RR de tipo RRSIG.

Para un servidor dado, el conjunto de llaves zsk y ksk se encuentra definido por el siguiente Record:

$$\textbf{keys} \stackrel{\text{def}}{=} [\![\ \textbf{zsk} : keySet,\ \textbf{ksk} : keySet\ ]\!]$$

4.2.5. Delegaciones

Conceptualmente, una delegación describe una relación Padre-Hijo entre servidores DNSSEC, y es el mecanismo esencial de construcción de la cadena de confianza de DNSSEC. Una delegación queda definida con el siguiente Record:

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

donde **Tsrv** indica el servidor "Hijo" a quién está delegando, y **rrDS** no es más que un RR seguro (el cual contiene un RR de tipo DS y su respectiva firma RRSIG).

Un servidor dado puede estar asociado a una o más delegaciones. Esta relación está definida mediante una función $DSpDB$, la cual está indexada por un servidor (*Process*) al cuál se delega la autoridad, y en su rango se encontrará una delegación propiamente dicha (DSp):

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

4.2.6. El estado

El estado global del sistema se describe utilizando el siguiente tipo record

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

donde ***Servers*** es el conjunto de servidores DNS; ***ServerKeys*** es una función entre un servidor y su conjunto de llaves ZSK/KSK; ***TrustedServers*** son aquellos servidores conocidos como confiables y preconfigurados como tal de manera fuera de banda; ***Delegations*** es la función entre un servidor y sus correspondientes delegaciones; ***Parents*** es una función que relaciona un servidor dado con su respectivo "padre" (aquel que tiene el respectivo RR de tipo DS apuntando a él); la vista que un servidor tiene sobre la base de datos puede particionarse en una parte autoritativa (***viewAuth***) y otra de caché (***viewCache***); ***PendingQueries*** define la función entre un servidor y las solicitudes de resolución de nombres que ya ha realizado y para las cuales aún no tiene respuesta; ***SendBuffer*** son aquellas acciones de envío de respuestas y solicitudes que un servidor dado tendrá que ejecutar en cuanto sea posible (se utiliza este buffer de acciones para modelizar ciertas ejecuciones esperadas al producirse ciertas condiciones que veremos más adelante).

4.2.7. Observadores del estado

Los observadores del estado son predicados que abstraen otros mas complejos mejorando la legibilidad y comprensión de la especificación.

El cuadro 4.1 enumera todos los observadores definidos junto con sus signaturas.

El predicado ***isServer*** describe si un servidor dado pertenece al conjunto de servidores del sistema; ***isZSK*** e ***isKSK*** indican si una llave dada es efectivamente de tipo ZSK o KSK respectivamente; ***emptyIdx*** verifica si existe RRset para un índice en el dominio de la vista autoritativa de un servidor dado, ***isTimeout*** informa si un RR ha expirado (ya sea por el tiempo de vida del RR en sí o debido a la expiración de su correspondiente firma), ***verifySign*** chequea si una llave efectivamente ha firmado un RRset mediante un RRSIG dado, ***signedView*** describe si una vista (autoritativa o de caché) de un servidor dado se encuentra firmada correctamente por su llave de zona ZSK, ***checkDigest*** es utilizado para verificar que un servidor y su correspondiente llave KSK concuerden con el cálculo de un digest dado; ***sec_entry_point*** verifica que un servidor dado se encuentre

isServer : $State \to Process \to Prop$
isZSK : $RR \to Prop$
isKSK : $RR \to Prop$
emptyIdx : $State \to Process \to Idx \to Prop$
isTimeout : $State \to Process \to Idx \to RR \to Prop$
verifySign : $RR \to RRset \to RR \to Prop$
signedView : $Process \to DbMap \to keys \to Prop$
checkDigest : $State \to Process \to RR \to RR \to Prop$
sec_entry_point : $State \to Process \to Prop$
calculate_sign : $RR \to RRset \to RR$
calculate_DS : $Process \to RR \to RR$
no_match : $State \to Process \to DNSMessage \to bool$
makeMsg : $State \to Process \to QResult \to DNSMessage \to DNSMessage$
makeResp : $Process \to Process \to DNSMessage \to InfoMsg$
nodata_newrefer : $DNSMessage \to bool$
next : $RRset \to Process$

Cuadro 4.1: Observadores del estado

dentro de los servidores confiables ($TrustedServers$); ***no_match*** informa si dentro de un servidor dado, el RR de una solicitud recibida se encuentra dentro de su vista autoritativa o de caché; y finalmente ***nodata_newrefer*** nos indica si un mensaje contiene datos de respuesta a una solicitud hecha.

La función ***calculate_sign*** computa una firma RRSIG a partir de un RRset y una llave de zona dada, así como ***calculate_DS*** computa un digest DS para un servidor y una llave KSK dada.

La función ***makeMsg*** arma un paquete DNSMessage con su nuevo estado $NOMATCH$ o $RRFOUND$ para indicar si no se encontró el registro buscado (pudiendo hacer una redirección a un servidor "más cercano") o si el RR buscado fue encontrado. La función ***makeResp*** arma un nuevo paquete $InfoMsg$ pendiente para envío, informando los servidores "de" y "para" y el mensaje a transmitir; y finalmente ***next*** calcula un servidor "más cercano" (es decir con más probabilidades de tener una respuesta) al RRset buscado.

4.2.8. Propiedades de validez del estado

No todo elemento de tipo $State$ es un estado válido, para ello es necesario que sus componentes verifiquen ciertas relaciones. Para caracterizarlas formalmente se definen las siguientes propiedades de validez del estado, para un elemento $s : State$.

"Todo RRset dentro de la vista de un servidor dado, debe encontrarse firmado por su correspondiente firma RRSIG"

$$\textbf{RR\_integrity}\ s \stackrel{\text{def}}{=} \forall\ srv{:}Process,\ srv \in s.Servers \rightarrow signedView\ s\ srv\ (s.viewAuth\ srv) \wedge\ signedView\ s\ srv\ (s.viewCache\ srv)$$

"El conjunto de llaves DNSKEY de todo servidor debe estar firmado por su correspondiente llave KSK."

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

"Todo servidor debe, o bien ser conocido públicamente como confiable, o ser verificado por su correspondiente padre."

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

Finalmente se dice que un estado s es válido si cumple la siguiente propiedad:

$$\begin{aligned}\textbf{Valid}\ s \stackrel{\text{def}}{=} RR\_integrity\ s\\ \wedge ZSK\_integrity\ s\\ \wedge KSK\_integrity\ s\end{aligned}$$

4.3. Comandos

La interacción con el sistema se realiza mediante la ejecución de comandos. Estos especifican como evoluciona el estado y la respuesta del sistema. En esta sección se presenta su sintaxis, semántica y ejecución.

4.3.1. Sintaxis

La sintaxis y signatura de cada comando está definida por el conjunto inductivo no recursivo $Funcs$:

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

Las posibles respuestas del sistema están definidas mediante el tipo inductivo no recursivo $answer$:

$$
\begin{aligned}
\textbf{\textit{answer}} \stackrel{\text{def}}{=}\ & ok : answer \\
\mid\ & error : ErrorCode \to answer
\end{aligned}
$$

Todos los comandos al tener éxito en su ejecución devuelven *ok* como respuesta.

4.3.2. Semántica

La semántica de los comandos está especificada mediante pre y pos condiciones. Para cada comando, su precondición es un predicado sobre el estado y su postcondición es un predicado que relaciona el estado anterior y el posterior a la ejecución del mismo.

A continuación se procede a describir cada comando, detallando y formalizando sus respectivas pre y pos condiciones. En la especificación de las poscondiciones se han omitido todos los campos del estado que permanecen invariantes al ejecutar el comando en cuestión.

- ***Add_Server srv sk ds parent vA vC*** crea un nuevo servidor (srv) en el sistema de dominios, el cual ya debe haber sido configurado fuera de banda, haciéndolo consistente dentro del sistema. Para que la operación tenga éxito dicho servidor no debe existir previamente en el sistema, este nuevo servidor o bien debe ser un servidor conocido como confiable o bien debe estar firmado mediante un digest de

su respectivo padre (*parent*), así como también deberá verificar con sus propias delegaciones (*ds*) todos sus servidores hijos. Así mismo deberán verificarse todos los registros dentro de sus vistas autoritativas y de cache (vA y vC) mediante su llave local zsk (componente zsk dentro de *sk*).

Pre s (Add_Server srv sk ds parent vA vC) $\stackrel{\text{def}}{=}$

$$\begin{aligned}
&\neg\ isServer\ s\ srv \\
&\wedge\ (checkDigest\ s\ srv\ parent.rrDS.RR'\ sk.ksk.key \\
&\quad \vee\ sec_entry_point\ s\ srv) \\
&\wedge\ \forall\ (srv' : Process),\ srv' = (ds\ srv).Tsrv \rightarrow \\
&\quad checkDigest\ s\ srv'\ (ds\ srv).rrDS.RR'\ (s.ServerKeys\ srv').ksk.key \\
&\wedge\ signedView'\ srv\ vA\ sk \\
&\wedge\ signedView'\ srv\ vC\ sk
\end{aligned}$$

Pos s s' (Add_Server srv sk ds parent vA vC) $\stackrel{\text{def}}{=}$

$$\begin{aligned}
&s'.Servers = \{srv\}\ \cup\ s.Servers \\
&\wedge\ s'.ServerKeys\ srv = sk \\
&\wedge\ s'.viewAuth\ srv = vA \\
&\wedge\ s'.viewCache\ srv = vC \\
&\wedge\ s'.Delegations\ srv = ds \\
&\wedge\ s'.Parents\ srv = parent
\end{aligned}$$

- ***Delete_Server srv*** elimina un servidor del sistema, así como también son eliminadas todas las llaves, vistas, solicitudes pendientes, envíos pendientes, delegaciones y padres establecidos que este posee. Para que el comando se ejecute exitosamente el servidor debe existir en el sistema.

Pre s (Delete_Server srv) $\stackrel{\text{def}}{=}\ isServer\ s\ srv$

Pos s s' (Delete_Server srv) $\stackrel{\text{def}}{=}$

$$\begin{aligned}
&s'.Servers = s.Servers \setminus \{srv\} \\
&\wedge\ s'.ServerKeys\ srv = nullKey \\
&\wedge\ s'.viewAuth\ srv = emptyView \\
&\wedge\ s'.viewCache\ srv = emptyView \\
&\wedge\ s'.PendingQueries\ srv = nil \\
&\wedge\ s'.SendBuffer\ srv = emptySendQR \\
&\wedge\ s'.Delegations\ srv = emptyDSpDB \\
&\wedge\ s'.Parents\ srv = emptyDSp
\end{aligned}$$

- ***Add_RRset srv i rrSet*** inserta un nuevo RRset en la vista autoritativa de un servidor determinado del sistema. Para que el comando tenga éxito, el servidor debe existir y no debe poseer un RRset previo en el índice (i) especificado.

$$\textbf{\textit{Pre s (Add\_RRset srv i rrSet)}} \stackrel{\text{def}}{=} isServer\ s\ srv \\ \wedge\ emptyIdx\ s\ srv\ i$$

$$\textbf{\textit{Pos s s' (Add\_RRset srv i rrSet)}} \stackrel{\text{def}}{=} \\ ((s'.viewAuth\ srv)\ i).RRset' = rrSet \\ \wedge\ ((s'.viewAuth\ srv)\ i).RRsign = \\ calculate\_sign\ (s.ServerKeys\ srv).zsk.key\ rrSet$$

- ***Delete_RRset srv i*** elimina un RRset de la vista autoritativa de un servidor dado del sistema. Es necesario que el servidor exista, así como también esta vista debe poseer un RRset no vacío para el índice i dado, para que el comando tenga éxito.

$$\textbf{\textit{Pre s (Delete\_RRset srv i)}} \stackrel{\text{def}}{=} isServer\ s\ srv \\ \wedge\ emptyIdx\ s\ srv\ i$$

$$\textbf{\textit{Pos s s' (Delete\_RRset srv i)}} \stackrel{\text{def}}{=} \\ (s'.viewAuth\ srv)\ i = emptySecRRset$$

- ***Server_ZSK_rollover srv rrzsk*** realiza el rollover de la llave ZSK para un servidor dado del sistema. Para que el comando se ejecute exitosamente el servidor srv debe existir dentro del sistema y la llave $rrzsk$ debe ser efectivamente una llave de tipo ZSK, esto es una llave cuyo RRDATA posee la "Zone key flag" seteada pero no el "bit SEP".

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

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

- ***Server_KSK_rollover srv rrksk ksksign*** realiza el rollover de la llave KSK para un servidor dado del sistema. Para que el comando se ejecute exitosamente el servidor srv debe existir dentro del sistema y la llave $rrksk$ debe ser efectivamente una llave de tipo KSK, esto es una llave cuyo RRDATA posee la bandera "Zone key" y el "bit SEP" seteados.

  Pre s (Server_KSK_rollover srv rrksk ksksign) $\stackrel{\text{def}}{=}$

  $$\begin{aligned} & isServer\ s\ srv \\ \wedge\ & isKSK\ rrksk \\ \wedge\ & verifySign\ rrksk\ ((s.ServerKeys\ srv).zsk.key)\ ksksign \end{aligned}$$

  Pos s s' (Server_KSK_rollover srv rrksk ksksign) $\stackrel{\text{def}}{=}$

  $$\begin{aligned} & s'.ServerKeys\ (s'.Parents\ srv).Tsrv = \\ & \qquad s.ServerKeys\ (s.Parents\ srv).Tsrv \\ \wedge\ & (s'.ServerKeys\ srv).ksk.key = rrksk \\ \wedge\ & (s'.ServerKeys\ srv).ksk.ksign = ksksign \\ \wedge\ & (s'.ServerKeys\ srv).zsk.key = (s.ServerKeys\ srv).zsk.key \\ \wedge\ & (s'.ServerKeys\ srv).zsk.ksign = \\ & \qquad calculate\_sign\ \ rrksk \\ & \qquad\qquad (\{rrksk\} \cup\ \{(s.ServerKeys\ srv).zsk.key\}) \\ \wedge\ & if\ (srv\ \notin\ s.TrustedServers)\ then \\ & \quad ((s'.Delegations\ (s.Parents\ srv).Tsrv)\ srv).Tsrv = srv \\ & \quad \wedge\ ((s'.Delegations\ (s.Parents\ srv).Tsrv)\ srv).rrDS.RR' = \\ & \qquad calculate\_DS\ srv\ rrksk \\ & \quad \wedge\ ((s'.Delegations\ (s.Parents\ srv).Tsrv)\ srv).rrDS.Rsign = \\ & \qquad calculate\_sign \\ & \qquad\qquad (s.ServerKeys\ (s.Parents\ srv).Tsrv).zsk.key \\ & \qquad\qquad \{calculate\_DS\ srv\ rrksk\} \end{aligned}$$

- ***Add_TrustedServer srv*** estaría significando la publicación de un servidor como confiable (e.g. servidor raíz "."), el cuál todo servidor dentro del sistema puede configurar fuera de banda y considerarlo confiable. Para que el comando tenga éxito dicho servidor tiene que estar activo dentro del sistema y todavía no debe pertenecer al conjunto de servidores confiables.

  Pre s (Add_TrustedServer srv) $\stackrel{\text{def}}{=}\ srv\ \notin\ s.TrustedServers$

  Pos s s' (Add_TrustedServer srv) $\stackrel{\text{def}}{=}$

  $$s'.TrustedServers = \{srv\}\ \cup\ s.TrustedServers$$

- ***Del_TrustedServer srv*** representa un servidor que antes era conocido como confiable y que ahora ya no debe considerarse como tal. Para que la operación se ejecute exitosamente el servidor srv ya debe pertenecer al conjunto de servidores confiables.

$$\textbf{\textit{Pre s (Del\_TrustedServer srv)}} \stackrel{\text{def}}{=} isServer\ s\ srv \\ \wedge\ srv\ \in\ s.TrustedServers$$

$$\textbf{\textit{Pos s s' (Del\_TrustedServer srv)}} \stackrel{\text{def}}{=} \\ s'.TrustedServers = s.TrustedServers\ \setminus\ \{srv\}$$

- ***Send_Query from to m*** envía una solicitud (m) realizada por el servidor $from$ al servidor to. Para que el comando tenga éxito, los servidores $from$ y to deben pertenecer al conjunto de servidores reconocidos en el sistema, el RR consultado (encontrado en el campo Query del mensaje DNS m) no debe pertenecer a la vista autoritativa o de caché. Cabe aclarar que en el presente trabajo, como presunción, las consultas sólo pueden hacerse a servidores confiables (e.g. root ".").

$$\textbf{\textit{Pre s (Send\_Query from to m)}} \stackrel{\text{def}}{=} \\ isServer\ s\ from \\ \wedge\ isServer\ s\ to \\ \wedge\ ((s.viewAuth\ from)\ m.Q).RRset' = nil \\ \wedge\ ((s.viewCache\ from)\ m.Q).RRset' = nil \\ \wedge\ to \in\ s.TrustedServers$$

$$\textbf{\textit{Pos s s' (Send\_Query from to m)}} \stackrel{\text{def}}{=} \\ (s'.PendingQueries\ from)\ = \{m.Hdr\} \cup\ (s.PendingQueries\ from)$$

- ***Send_PendingQ srv*** Envía solicitudes pendientes. Es necesario para que el comando tenga éxito que el servidor srv se encuentre dentro del conjunto de servidores del sistema y que exista al menos una solicitud pendiente para el envío.

$$\textbf{\textit{Pre s (Send\_PendingQ srv)}} \stackrel{\text{def}}{=} isServer\ s\ from \\ \wedge\ (s.SendBuffer\ srv).SendQ\ \neq\ nil$$

$$\textbf{\textit{Pos s s' (Send\_PendingQ srv)}} \stackrel{\text{def}}{=} \\ (s'.SendBuffer\ srv).SendQ\ = tail\ (s.SendBuffer\ srv).SendQ$$

- ***Send_PendingR srv*** Envía respuestas pendientes. Es necesario para que el comando tenga éxito que el servidor srv se encuentre dentro del conjunto de servidores del sistema y que exista al menos una respuesta pendiente para el envío.

$$\textbf{\textit{Pre s (Send\_PendingR srv)}} \stackrel{\text{def}}{=} isServer\ s\ from \\ \wedge\ (s.SendBuffer\ srv).SendR \neq nil$$

$$\textbf{\textit{Pos s s' (Send\_PendingR srv)}} \stackrel{\text{def}}{=} \\ (s'.SendBuffer\ srv).SendR = tail\ (s.SendBuffer\ srv).SendR$$

- ***Receive_Query from to m*** procesa una solicitud recibida y dependiendo el caso enviará una respuesta con los datos solicitados, o bien una redirección a un servidor más cercano. Para que el comando tenga éxito los servidores $from$ y to deben existir en el sistema.

$$\textbf{\textit{Pre s (Receive\_Query from to m)}} \stackrel{\text{def}}{=} isServer\ s\ from \\ \wedge\ isServer\ s\ to$$

$$\textbf{\textit{Pos s s' (Receive\_Query from to m)}} \stackrel{\text{def}}{=} \\ if\ (no\_match\ s\ to\ m)\ then \\ (s'.SendBuffer\ to).SendR = \\ \{\langle\ to,\ from,\ (makeMsg\ s\ to\ NOMATCH\ m)\ \rangle\} \\ \cup\ (s.SendBuffer\ to).SendR \\ else \\ (s'.SendBuffer\ to).SendR = \\ \{\langle\ to,\ from,\ (makeMsg\ s\ to\ RRFOUND\ m)\ \rangle\} \\ \cup\ (s.SendBuffer\ to).SendR$$

- ***Receive_Response from to m*** procesa una respuesta m recibida. Este es un comando crucial para la formalización del sistema, ya que en este se evalúa si aceptar o no un RR en la vista de caché de un servidor dado, lo cual evitaría el envenenamiento de caché que discutimos en el capítulo 2. Para que la operación tenga éxito los servidores $from$ y to deben pertenecer al conjunto de servidores del sistema, y la respuesta recibida debe corresponderse a una respuesta esperada, es decir, a una

solicitud previamente hecha.

$$\textbf{\textit{Pre s (Receive\_Response from to m)}} \stackrel{\text{def}}{=} isServer\ s\ from \\ \wedge\ isServer\ s\ to \\ \wedge\ m.Hdr\ \in\ (s.PendingQueries\ to)$$

$$\textbf{\textit{Pos s s' (Receive\_Response from to m)}} \stackrel{\text{def}}{=} \\ if\ (nodata\_newrefer\ m)\ then \\ (s'.SendBuffer\ to).SendQ = \\ \{(makeResp\ to\ (next\ m.Additional.RRset')\ m)\} \\ \cup\ (s.SendBufferto).SendQ \\ elseif\ (verifySign\ (s.ServerKeys\ to).zsk.key \\ m.Ans.RRset'\ \ m.Ans.RRsign)\ then \\ ((s'.viewCache\ to)\ m.Q).RRset' = m.Ans.RRset' \\ \wedge\ ((s'.viewCache\ to)\ m.Q).RRsign = m.Ans.RRsign \\ \wedge\ s'.PendingQueries\ to = (s.PendingQueries\ to) \setminus \{m.Hdr\}$$

- ***RR_TimeOut srv i rr*** se ejecuta al expirar el tiempo de validez de un RR para un servidor dado. Para que la operación se ejecute exitosamente, el servidor *srv* debe pertenecer al conjunto de servidores del sistema, y el RR en cuestión debe haber expirado ya sea por timeout del RR propiamente dicho o por su correspondiente firma RRSIG.

$$\textbf{\textit{Pre s (RR\_TimeOut srv i rr)}} \stackrel{\text{def}}{=} isServer\ s\ from \\ \wedge\ isTimeout\ s\ srv\ i\ rr \\ \wedge (rr\ \in\ (s.viewCache\ srv\ i).RRset' \\ \vee\ rr\ \in\ (s.viewAuth\ srv\ i).RRset')$$

$$\textbf{\textit{Pos s s' (RR\_TimeOut srv i rr)}} \stackrel{\text{def}}{=} \\ if\ (rr\ \in\ (s.viewAuth\ srv\ i).RRset')\ then \\ ((s'.viewAuth\ srv)\ i).RRset' = ((s.viewAuth\ srv)\ i).RRset' \setminus \{rr\} \\ \wedge\ ((s'.viewAuth\ srv)\ i).RRsign = \\ calculate\_sign\ (s.ServerKeys\ srv).zsk.key \\ ((s'.viewAuth\ srv)\ i).RRset' \\ else \\ ((s'.viewCache\ srv)\ i).RRset' = ((s.viewCache\ srv)\ i).RRset' \setminus \{rr\} \\ \wedge\ ((s'.viewCache\ srv)\ i).RRsign = \\ calculate\_sign\ (s.ServerKeys\ srv).zsk.key \\ ((s'.viewCache\ srv)\ i).RRset'$$

4.3.3. Errores

Cuando se intenta ejecutar un comando y alguna de las precondiciones del mismo no es válida, el sistema responde con un código de error. Los posibles códigos de error quedan definidos mediante el siguiente tipo enumerado:

$$\begin{aligned}\textbf{\textit{ErrorCode}} \stackrel{\text{def}}{=} & \; server\_exists \mid server\_not\_exists \mid invalid\_sign \\ & \mid bad\_digest \mid corrupt\_viewAuth \mid known\_anwer \\ & \mid rrset\_exists \mid rrset\_not\_exists \mid invalid\_zsk \\ & \mid server\_out\_of\_trusted\_chain \mid invalid\_ksk \\ & \mid corrupt\_viewCache \mid server\_not\_trusted \\ & \mid empty\_SendQ\_buffer \mid empty\_SendR\_buffer \\ & \mid query\_not\_asked \mid rr\_not\_timeout\end{aligned}$$

Para cada comando f se especifica el cuadro 4.2 que asocia el código de error y la condición del comando (la negación de la precondición P_f) definiendo la relación $ErrorMsg_f$ entre estados válidos del sistema y mensajes de error.

4.3.4. Ejecución de los comandos

La ejecución de un comando f en un estado s produce un nuevo estado s' y una respuesta r donde la relación entre el estado previo, el nuevo y la respuesta, está determinada por la relación de postcondición Pos_f.

$$\textbf{\textit{exec}} : State \to Funcs \to answer \to State \to Prop$$

$$\textbf{\textit{exec\_pre}} \; \frac{Pre\ s\ f \qquad Pos\ s\ s'\ f \qquad ans = ok}{exec\ s\ f\ ans\ s'}$$

$$\textbf{\textit{exec\_npre}} \; \frac{\neg\ (Pre\ s\ f) \qquad \exists\ ec{:}ErrorCode,\ s = s' \wedge ErrorMsg\ s\ f\ ec \wedge ans = error\ ec}{exec\ s\ f\ ans\ s'}$$

Si la precondición $Pre\ s\ f$ se cumple, entonces el estado resultante s' y la respuesta ans son las descriptas por la relación $exec$. En cambio, si $Pre\ s\ f$ no se cumple, entonces el estado s no cambia y la respuesta del sistema es el mensaje de error determinado por la relación $ErrorMsg$.

Cuadro 4.2: Cuadro de códigos de error

Add_Server srv sk ds parent vA vC	
$isServer\ s\ srv$	server_exists
$\neg\ checkDigest\ s\ srv\ parent.(rrDS).(RR')\ sk.(ksk).(key)$ $\wedge\ \neg\ sec\_entry\_point\ s\ srv$	server_out_of_trusted_chain
$\exists\ srv' \in\ dom\ ds \rightarrow$ $\neg\ checkDigest\ s\ srv'(ds\ srv).rrDS.RR'$ $(s.ServerKeys\ srv').ksk.key$	bad_digest
$\neg\ signedView'\ srv\ vA\ sk$	corrupt_viewAuth
$\neg\ signedView'\ srv\ vC\ sk$	corrupt_viewCache
Delete_Server srv	
$\neg\ isServer\ s\ srv$	server_not_exists
Add_RRset srv i rrSet	
$\neg\ isServer\ s\ srv$	server_not_exists
$\neg\ emptyIdx\ s\ srv\ i$	rrset_exists
Delete_RRset srv i	
$\neg\ isServer\ s\ srv$	server_not_exists
$emptyIdx\ s\ srv\ i$	rrset_exists
Server_ZSK_rollover srv rrzsk	
$\neg\ isServer\ s\ srv$	server_not_exists
$\neg\ isZSK\ rrzsk$	invalid_zsk
Server_KSK_rollover srv rrksk ksksign	
$\neg\ isServer\ s\ srv$	server_not_exists
$\neg\ isKSK\ rrksk$	invalid_ksk
$\neg\ verifySign\ rrksk$ $\{(s.ServerKeys\ srv).zsk.key\}\ ksksign$	invalid_sign
Add_TrustedServer srv	
$srv\ \in\ s.TrustedServers$	server_already_trusted
Del_TrustedServer srv	
$srv\ \notin\ s.TrustedServers$	server_not_trusted
Send_Query from to m	
$\neg\ isServer\ s\ from \vee\ \neg\ isServer\ s\ to$	server_not_exists
$((s.viewAuth\ from)\ m.Q).RRset' \neq\ nil$ $\vee\ ((s.viewCache\ from)\ m.Q).RRset' \neq\ nil)$	known_anwer
$to\ \notin\ s.TrustedServers$	server_not_trusted
Send_PendingQ srv	
$\neg\ isServer\ s\ srv$	server_not_exists
$(s.SendBuffer\ srv).SendQ = nil$	empty_SendQ_buffer
Send_PendingR srv	
$\neg\ isServer\ s\ srv$	server_not_exists
$(s.SendBuffer\ srv).SendR = nil$	empty_SendR_buffer

Receive_Query from to m	
$\neg\ isServer\ s\ from \vee\ \neg\ isServer\ s\ to$	server_not_exists
Receive_Response from to m	
$\neg\ isServer\ s\ from \vee\ \neg\ isServer\ s\ to$	server_not_exists
$\neg\ m.Hdr\ \in\ s.PendingQueries\ to$	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

Capítulo 5

Verificación

En el capítulo 4 se definió la formalización de DNSSEC en el CCI. En el presente capítulo se desarrolla la verificación de la misma, dividida en dos etapas. En una primera etapa se demuestra la invariancia de la validez del estado respecto a la ejecución de los comandos (sección 5.1), es decir, se demuestra que la semántica dada a los comandos preservan las propiedades de validez del estado, garantizando fuertes mejoras de seguridad con respecto al modelo anterior, DNS. Y en una segunda etapa se demuestra muy brevemente una inconsistencia encontrada en el sistema (sección 5.2).

En esta sección se desarrollan las pruebas de las propiedades mas relevantes y de manera informal. Todos los resultados fueron verificados formalmente en la teoría desarrollada en COQ.

A lo largo de este capítulo se consideran definidos $s\ s' : State$ y además el estado s es considerado válido ($Valid\ s$).

5.1. Invariancia de la validez del estado

Para demostrar que el estado s' especificado por la relación de postcondición de cada comando es válido, es necesario mostrar que valen cada una de las propiedades que componen a $Valid\ s'$: $RR\_integrity\ s'$, $ZSK\_integrity\ s'$ y $KSK\_integrity\ s'$.

Las pruebas para los comandos que poseen poscondiciones que mantienen invariantes los campos del estado relacionados con alguna de estas propiedades son omitidas ya que se verifican trivialmente.

5.1.1. Hipótesis consideradas

Primero es necesario plantear algunas hipótesis que serán necesarias para el desarrollo posterior de las demostraciones de los invariantes.

En primer lugar, la siguiente hipótesis afirma que si un RRset está firmado mediante

la función *calculate_sign* y su correspondiente llave de zona, se verificará la propiedad $verifySign$. Formalmente:

$$\forall\ (zsk\_r\ sig\_r : RR)\ (rs : RRset),$$
$$calculate\_sign\ zsk\_r\ rs = sig\_r\ \rightarrow\ verifySign\ zsk\_r\ rs\ sig\_r \tag{5.1}$$

Similar a la anterior, la siguiente hipótesis sostiene que si un digest fue generado con la función *calculate_ds*, se verificará la propiedad *checkDigest*:

$$\forall\ (s : State)\ (srv : Process)\ (digest\ ksk : RR), \tag{5.2}$$
$$calculate\_DS\ srv\ ksk = digest\ \rightarrow$$
$$checkDigest\ s\ srv\ digest\ ksk$$

5.1.2. Transición válida para el comando Add_RRset

El siguiente lema prueba que la postcondición de *Add_RRset* especifica un estado s' donde se cumple *RR_integrity*.

Lema 5.1.1 *Sean* $(s\ s' : State)$, $(server : Process)$, $(id : Idx)$ *y* $(rr : RRset)$.
Si se cumplen:

$$Valid\ s$$
$$Pre\ s\ (Add\_RRset\ server\ id\ rr)$$
$$Pos\ s\ s'\ (Add\_RRset\ server\ id\ rr)$$

entonces en el estado s' *vale RR_integrity.*

Demostración. La operación *Add_RRset* sólo modifica el componente viewAuth del estado. De la postcondición de *Add_RRset* se conoce que

$$((s'.viewAuth\ srv)\ i).RRset' = rrSet \tag{5.3}$$

$$((s'.viewAuth\ srv)\ i).RRsign =$$
$$calculate\_sign\ (s.ServerKeys\ srv).zsk.key\ rrSet \tag{5.4}$$

Expandiendo la definición de $RR\_integrity\ s'$ se tiene

$$\forall\ srv : Process,$$
$$srv \in\ s'.Servers \rightarrow \tag{5.5}$$
$$signedView\ s'\ srv\ (s'.viewAuth\ srv) \tag{5.6}$$
$$\wedge\ signedView\ s'\ srv\ (s'.viewCache\ srv) \tag{5.7}$$

Como s es un estado válido y *Add_RRset* sólo modifica su componente *viewAuth*, sólo debemos verificar que valga (5.6). Más aún, dentro de viewAuth sólo se modifica la vista

para el índice i como podemos ver en (5.3) y (5.4). Expandiendo la definición de (5.6), bastará con demostrar:

$$\begin{aligned} &verifySign\ (s'.ServerKeys\ srv).zsk.key \\ &\qquad ((s'.viewAuth\ srv)\ i).RRset' \\ &\qquad ((s'.viewAuth\ srv)\ i).RRsign \end{aligned} \tag{5.8}$$

Utilizando la hipótesis (5.1), será suficiente probar:

$$\begin{aligned} &((s'.viewAuth\ srv)\ i).RRsign = \\ &\qquad calculate\_sign\ (s'.ServerKeys\ srv).zsk.key \\ &\qquad\qquad ((s'.viewAuth\ srv)\ i).RRset' \end{aligned} \tag{5.9}$$

Dado que el componente $ServerKeys$ del estado se mantiene igual, sabemos que:

$$(s'.ServerKeys\ srv).zsk.key = (s.ServerKeys\ srv).zsk.key \tag{5.10}$$

Reescribiendo 5.10 y 5.4 en 5.9 queda:

$$\begin{aligned} &((s'.viewAuth\ srv)\ i).RRsign = \\ &\qquad calculate\_sign\ (s.ServerKeys\ srv).zsk.key\ rrSet \end{aligned}$$

Lo cual es válido por (5.3).

□

Por lo tanto podemos afirmar que al agregarse un nuevo registro a la vista autoritativa de un servidor dado, se estará agregando su respectiva firma, manteniendo invariante la validez de dicha vista ($RR\_integrity$).

5.1.3. Transición válida para el comando Receive_Response

El siguiente lema prueba que la postcondición de *Receive_Response* especifica un estado s' donde se cumple $RR\_integrity$.

Lema 5.1.2 *Sean* $(s\ s' : State)$, $(srv\_from\ srv\_to : Process)$ *y* $(m' : DNSMessage)$. *Si se cumplen:*

$$\begin{aligned} &Valid\ s \\ &Pre\ s\ (Receive\_Response\ srv\_from\ srv\_to\ m') \\ &Pos\ s\ s'\ (Receive\_Response\ srv\_from\ srv\_to\ m') \end{aligned}$$

entonces en el estado s' *vale* $RR\_integrity$.

Demostración. En este caso para probar $RR\_integrity$ bastará con verificar (5.7). Comenzaremos la demostración haciendo el siguiente análisis por casos, primero se analiza si se recibe una respuesta con contenido autoritativo o no, y luego, si la respuesta efectivamente tiene contenido autoritativo, se analiza si se verifica la firma del RRset recibido. Más formalmente, de la postcondición del comando:

$$
\begin{aligned}
&if\ nodata\_newrefer\ m && (5.11)\\
&\quad then\ (s'.SendBuffer\ to).SendQ = \\
&\qquad \{(makeResp\ to\ (next\ m.Additional.RRset')\ m)\} \\
&\qquad \cup\ (s.SendBufferto).SendQ \\
&elseif\ (verifySign\ (s.ServerKeys\ to).zsk.key\ m.Ans.RRset'\ m.Ans.RRsign) && (5.12)\\
&\quad then\ ((s'.viewCache\ to)\ m.Q).RRset' = m.Ans.RRset' \\
&\qquad \wedge\ ((s'.viewCache\ to)\ m.Q).RRsign = m.Ans.RRsign \\
&\qquad \wedge\ s'.PendingQueries\ to = (s.PendingQueries\ to) \setminus \{m.Hdr\}
\end{aligned}
$$

Primero analizamos (5.11), si la respuesta recibida no contiene datos (i.e. $nodata\_newreferer\ m' = False$), entonces al ejecutar el comando $Receive\_Response$, el componente del estado $viewCache$ se mantendrá invariante, por lo cual (5.7) será válido.

En el caso en que la respuesta efectivamente contenga datos en su sección $Auth$, considerando ahora (5.12), habrá que analizar si se verifican el RRset recibido con su correspondiente firma RRsig. Si la firma no es verificada el componente $viewCache$ no se modificará por lo cual (5.7) será válido. Si se da el caso en el que la firma es verificada y añadida a la vista $viewCache$ del servidor to, esta será válida, ya que es condición para que el RRset y su respectiva firma sean agregados.

□

El lema recién demostrado tiene implicancias muy importantes, ya que es una situación crucial en la que utilizando el sistema DNS clásico se podría producir envenenamiento de caché. Sin embargo, vemos como utilizando $DNSSEC$, se utilizan las firmas de los $RRsets$ recibidos y analizar si la respuesta recibida será guardada en caché o no.

5.1.4. Transición válida para el comando Server_ZSK_rollover

El siguiente lema prueba que la postcondición de $Server\_ZSK\_rollover\ srv\ rrzsk$ especifica un estado s' donde se cumple $RR\_integrity$.

Lema 5.1.3 *Sean* $(s\ s' : State)$, $(srv : Process)$ *y* $(rrzsk : RR)$.
Si se cumplen:

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

entonces en el estado s' *vale* $RR\_integrity$.

Demostración. De la postcondición de $Server\_ZSK\_rollover$ se conoce que :

$$(s'.ServerKeys\ srv).zsk.key = rrzsk \tag{5.13}$$
$$(s'.ServerKeys\ srv).ksk.key = (s.ServerKeys\ srv).ksk.key \tag{5.14}$$

$$(s'.ServerKeys\ srv).zsk.ksign = \tag{5.15}$$
$$calculate\_sign\ (s.ServerKeys\ srv).ksk.key$$
$$(\{(s.ServerKeys\ srv).ksk.key\} \cup \{rrzsk\})$$

$$\forall\ i{:}Idx,\ ((s'.viewAuth\ srv)\ i).RRsign = \tag{5.16}$$
$$calculate\_sign\ rrzsk\ (s.viewAuth\ srv\ i).RRset'$$

Ya que la vista de caché no es modificada por este comando, tendremos que demostrar (5.6). Expandiendo el objetivo de prueba se tiene

$$\forall\ i{:}Idx,\ (s'.viewAuth\ srv)\ i = emptySecRRset \tag{5.17}$$
$$\vee\ verifySign\ (s'.ServerKeys\ srv).zsk.key$$
$$((s'.viewAuth\ srv)\ i).RRset' \tag{5.18}$$
$$((s'.viewAuth\ srv)\ i).RRsign$$

Al realizar un rollover y firmar nuevamente la vista, si un determinado RRset se encontraba vacío en la vista (i.e. $(viewAuth\ s'\ srv)\ i = emptySecRRset$), este seguirá estándolo, resultando (5.17) trivial. Por este motivo, de la disyunción planteada, la demostración se centrará en probar (5.18), para esto se utiliza la hipótesis (5.1), reescribiendo el objetivo queda:

$$\forall\ i{:}Idx,\ ((s'.viewAuth\ srv)\ i).RRsign =$$
$$calculate\_sign\ (s'.ServerKeys\ srv).zsk.key \tag{5.19}$$
$$((s'.viewAuth\ srv)\ i).RRset'$$

Reescribiendo (5.13) en (5.19), obtenemos (5.16), lo cuál es válido por hipótesis.

□

A continuación se deberá demostrar que para el mismo comando $Server\_ZSK\_rollover$, si se cumplen sus pre y pos condiciones también se verificará $ZSK\_integrity$ en el estado s'.

Demostración.

Expandiendo la definición de $ZSK\_integrity\ s'$ se tiene:

$$\forall\ srv{:}Process,\ srv \in s'.Servers \rightarrow \tag{5.20}$$
$$verifySign\ (s'.ServerKeys\ srv).ksk.key$$
$$(\{(s'.ServerKeys\ srv).ksk.key\} \cup \{(s'.ServerKeys\ srv).zsk.key\})$$
$$(s'.ServerKeys\ srv).zsk.ksign$$

Al reescribir (5.1) en (5.20) se obtiene el siguiente objetivo de prueba:

$$\forall\ srv{:}Process,\ srv\ \in\ s'.Servers\ \rightarrow \tag{5.21}$$
$$(s'.ServerKeys\ srv).zsk.ksign =$$
$$calculate\_sign\ (s'.ServerKeys\ srv).ksk.key$$
$$(\{(s'.ServerKeys\ srv).ksk.key\}$$
$$\cup\ \{(s'.ServerKeys\ srv).zsk.key\}) \tag{5.22}$$

Al reemplazar (5.13) y (5.14), se obtiene (5.15), por lo cual podemos concluir que nuestro planteo inicial es efectivamente cierto.

□

Al hacerse un rollover de la llave de zona de un servidor dado, se vuelven a generar las firmas de todos los *RRsets* de la zona, inclusive de la propia llave *zsk* recién introducida, manteniendo consistente la vista autoritativa del servidor en cuestión. Esto se ve reflejado en los lemas recién demostrados.

5.1.5. Transición válida para el comando Server_KSK_rollover

El siguiente lema prueba que la postcondición de $(Server\_KSK\_rollover\ srv\ rrksk\ ksksign)$ especifica un estado s' donde se cumple $ZSK\_integrity$.

Lema 5.1.4 *Sean* $(s\ s'{:}State)$, $(srv{:}Process)$ *y* $(rrksk\ ksksign{:}RR)$.
Si se cumplen:

$$Valid\ s$$
$$Pre\ s\ (Server\_KSK\_rollover\ srv\ rrksk\ ksksign)$$
$$Pos\ s\ s'\ (Server\_KSK\_rollover\ srv\ rrksk\ ksksign)$$

entonces en el estado s' *vale* $ZSK\_integrity$.

Demostración. De la postcondición de $Server\_KSK\_rollover\ srv\ rrksk\ ksksign$ se conoce que:

$$(s'.ServerKeys\ srv).ksk.key = rrksk \tag{5.23}$$
$$(s'.ServerKeys\ srv).ksk.ksign = ksksign \tag{5.24}$$

$$(s'.ServerKeys\ srv).zsk.key = (s.ServerKeys\ srv).zsk.key \tag{5.25}$$
$$(s'.ServerKeys\ srv).zsk.ksign = \tag{5.26}$$
$$calculate\_sign\ rrksk$$
$$(\{rrksk\} \cup \{(s.ServerKeys\ srv).zsk.key\})$$

$$if\ srv \notin s.TrustedServers\ then \tag{5.27}$$
$$(s'.Parents\ srv).rrDS.RR' = calculate\_DS\ srv\ rrksk \tag{5.28}$$

y expandiendo el objetivo de prueba se tiene:

$$\forall\ srv{:}Process,\ srv \in s'.Servers \rightarrow \tag{5.29}$$
$$verifySign\ (s'.ServerKeys\ srv).ksk.key$$
$$(\{(s'.ServerKeys\ srv).ksk.key\} \cup \{(s'.ServerKeys\ srv).zsk.key\})$$
$$(s'.ServerKeys\ srv).zsk.ksign$$

utilizando la hipótesis (5.1), y reescribiendo (5.23) y (5.25) en (5.29), obtenemos (5.26), y al ser ésta una hipótesis conocida queda probado el lema actual.

□

Al demostrar este lema se está verificando que al realizar un rollover de la llave *ksk* se generan nuevas firmas para la llave *zsk* manteniendo a esta última consistente.

Lema 5.1.5 *Para el mismo comando, $Server\_KSK\_rollover$, si se cumplen sus pre y pos condiciones también se verificará $KSK\_integrity$ en el estado s'.*

Demostración. Expandiendo la definición de $KSK\_integrity\ s'$ se tiene:

$$\forall\ srvH{:}Process,\ srvH \in s'.TrustedServers \rightarrow \tag{5.30}$$
$$\vee\ checkDigest\ s'\ srvH \tag{5.31}$$
$$(s'.Parents\ srvH).rrDS.RR'$$
$$(s'.ServerKeys\ srvH).ksk.key$$

El comando $Server\_KSK\_rollover$ no modifica el componente del estado $TrustedServers$, por lo tanto si el servidor en cuestión (srv) se encuentra dentro del conjunto de servidores

confiables la demostración se resuelve trivialmente. A continuación se analiza el término (5.31) de nuestro objetivo, quedando:

$$\begin{aligned} &checkDigest\ s'\ srv \\ &\qquad (s'.Parents\ srv).rrDS.RR' \\ &\qquad (s'.ServerKeys\ srv).ksk.key \end{aligned} \tag{5.32}$$

Según lo planteado anteriormente ($srv \notin s.TrustedServers$) se cumple (5.27). Por lo tanto podemos reemplazar (5.23) y (5.28) en (5.30), obteniendo el siguiente objetivo de prueba:

$$checkDigest\ s'\ srv\ (calculate\_DS\ srv\ rrksk)\ rrksk \tag{5.33}$$

Aplicando la hipótesis (5.2) se obtiene una tautología, demostrando el lema.

□

A partir de la verificación de este último lema podemos confirmar que al realizar un rollover de la llave *ksk* se renueva el record *DS* ubicado en el servidor padre, y manteniendo segura la cadena de confianza. Si bien aquí vemos lo que debiera suceder según la especificación propuesta, no se encuentra especificada una implementación oficial para realizar esta comunicación con el padre para realizar dicha actualización.

5.1.6. Transición válida para el comando TimeOut

El siguiente lema demuestra que la validez del estado se mantiene invariante al ejecutar el comando *TimeOut*, especificando un estado s' donde se cumple *RR_integrity*.

Lema 5.1.6 *Sean* $(s\ s' : State)$, $(srv : Process)$, $(irset : Idx)$ *y* $(rrt : RR)$.
Si se cumplen:

$$\begin{aligned} &Valid\ s \\ &Pre\ s\ (RR\_TimeOut\ srv\ irset\ rrt) \\ &Pos\ s\ s'\ (RR\_TimeOut\ srv\ irset\ rrt) \end{aligned}$$

entonces en el estado s' *vale RR_integrity.*

Demostración. Para probar *RR_integrity* habrá que analizar si el RR que ha expirado pertenece a la vista de autoritativa, en tal caso habrá que verificar (5.6), o si pertenece a la vista caché, y en este otro caso habrá que probar (5.7).

Comenzaremos por demostrar (5.6). Para este caso las poscondiciones relevantes del comando son:

$$((s'.viewAuth\ srv)\ irset).RRset' = ((s.viewAuth\ srv)\ irset).RRset' \setminus \{rrt\} \tag{5.34}$$

$$\begin{aligned} ((s'.viewAuth\ srv)\ irset).RRsign = &\ calculate\_sign \\ &(s.ServerKeys\ srv).zsk.key \\ &((s'.viewAuth\ srv)\ irset).RRset' \end{aligned} \tag{5.35}$$

Suponiendo que $(s'.viewAuth\ srv)\ irset \neq \emptyset$, el objetivo de prueba queda:

$$\begin{aligned} verifySign\ &(s'.ServerKeys\ srv).zsk.key \\ &((s'.viewAuth\ srv)\ irset).RRset' \\ &((s'.viewAuth\ srv)\ irset).RRsign \end{aligned} \tag{5.36}$$

Sabiendo que $s'.ServerKeys = s.ServerKeys$ y reescribiendo (5.35) se tiene:

$$\begin{aligned} verifySign\ &(s.ServerKeys\ srv).zsk.key \\ &((s'.viewAuth\ srv)\ irset).RRset' \\ &(calculate\_sign \\ &\qquad (s.ServerKeys\ srv).zsk.key \\ &\qquad ((s'.viewAuth\ srv)\ irset).RRset') \end{aligned} \tag{5.37}$$

Reescribiendo la hipótesis (5.1) en el objetivo se obtiene una tautología, quedando demostrado que se cumple $RR\_integrity$ para s'.

Para el análisis del caso en que el RR que ha expirado pertenece a la vista de caché, el razonamiento será el mismo y la demostración será análoga.

□

En este caso se ve como al expirar un determinado $RRset$ la vista se mantiene válida, ya que al suceder esto se elimina tanto el $RRset$ como su firma.

5.1.7. Invariancia de la validez del estado

Finalmente, el siguiente teorema afirma que la semántica dada a los comandos mantiene la validez del estado. Esto garantiza que los comandos no pueden hacer transicionar al sistema a un estado que no represente un estado de DNSSEC.

Teorema 5.1.1 *Sean* $(s : State)$ *y* $(f : Func)$*. Si se cumple que* $Valid\ s$*,* $Pre\ s\ f$ *y* $Pos\ s\ s'\ f$ *entonces* $Valid\ s'$*.*

Demostración. La prueba procede realizando un análisis por casos de los comandos. Como se mencionó al comienzo del capítulo, se considera válida la invariancia de la validez del estado para aquellos comandos que poseen una postcondición que hace la prueba relativamente simple o trivial. Para el resto de los comandos, la misma queda probada por los lemas demostrados en esta sección.

□

5.2. Inconsistencia en la cadena de confianza

Como se demostró en la sección anterior, la invariancia en la validez del estado nos asegura que al ejecutar cualquier comando se obtiene un nuevo estado en el cual se cumplen las propiedades planteadas para mantener una cadena de confianza válida. Al hacer

un análisis sobre la operación $Server\_ZSK\_rollover$ se pudo encontrar una pequeña inconsistencia en los datos del sistema, la cuál al investigar trabajos anteriores hechos sobre $DNSSEC$, se ha encontrado que en [5] se llega a este mismo hallazgo.

Para que un RR sea quitado de una vista (ya sea autoritativa o de caché) sólo se verifica que se haya cumplido su TTL. El problema se produce cuando se realiza un *rollover* de una llave para la cual todavía no se ha cumplido su TTL (e.g. si se detecta que la llave de una zona se encuentra comprometida y es necesario cambiarla). Al realizar esta operación habrá que volver a firmar todos los $RRset$ de la zona para generar los nuevos RRs de tipo $RRSIG$, y como consecuencia, todos aquellos servidores que tengan registros en su vista de caché, adquiridos de la vista autoritativa del servidor en cuestión, quedarán inconsistentes con los verdaderos valores. Esto puede traer problemas de seguridad, ya que se podría producir envenenamiento de caché dentro del sistema.

Supongamos que el servidor $srvOldCache$ tenía información de srv en su vista de caché, digamos del índice "a''":

$$((s.viewAuth\ srv)\ a).RRsign = ((s.viewCache\ srvOldCache)\ a).RRsign \tag{5.38}$$

Lema 5.2.1 *Sean* $(s\ s' : State)$, $(rrzsk : RR)$, $(a : Idx)$ *y* $(srvsrvOldCache : Process)$. *Si se cumplen:*

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

Entonces, al realizar un rollover de la llave zsk de srv, se producirá la inconsistencia antes mencionada:

$$((s'.viewAuth\ srv)\ a).RRsign \neq\ ((s'.viewCache\ srvOldCache)\ a).RRsign \tag{5.39}$$

Demostración. Para la demostración se tendrá en cuenta la siguiente hipótesis:

$$\begin{aligned} \forall\ (srv : Process)\ (i : Idx)\ (zsk1\ zsk2 : RR), \quad & \\ zsk1\ \neq\ zsk2 \rightarrow & \\ calculate\_sign\ zsk1\ ((s.viewAuth\ srv)\ i).RRset' \neq & \\ calculate\_sign\ zsk2\ ((s.viewAuth\ srv)\ i).RRset' & \end{aligned} \tag{5.40}$$

la cual enuncia que si un mismo registro RR se firma con distintas llaves zsk se obtendrán distintos registros $RRSIG$.

De la postcondición del comando sabemos:

$$(s'.ServerKeys\ srv).zsk.key = rrzsk \tag{5.41}$$

$$\begin{aligned} \forall\ i : Idx,\ ((s'.viewAuth\ srv)\ i).RRsign = & \\ calculate\_sign\ rrzsk\ (s.viewAuth\ srv\ i).RRset' & \end{aligned} \tag{5.42}$$

Se realizará la demostración por el absurdo, partiendo de que en realidad si se cumple la propiedad planteada. Al reemplazar (5.38) y sabiendo que la vista de caché del sistema se mantiene invariante ($s'.viewCache = s.viewCache$), obtenemos el siguiente objetivo de prueba:

$$((s'.viewAuth\ srv)\ a).RRsign \neq\ ((s.viewAuth\ srv)\ a).RRsign \tag{5.43}$$

Al aplicar la definición de *calculate_sign* y la hipótesis planteada (5.40) llegamos a un absurdo. Esto nos permite concluir que (5.39) es válido.

□

Si bien se llega a la misma conclusión que en [5], en este trabajo en particular se desarrolla una especificación formal completa del modelo, en un formalismo que permite derivar prototipos funcionales certificados; y en este caso permite analizar y razonar sobre la validez de la cadena de confianza propuesta en DNSSEC.

Capítulo 6

Conclusiones

El presente capítulo desarrolla en secciones las decisiones tomadas durante la formalización en el cálculo de construcciones inductivas expuestas en el capítulo 4, conclusiones sobre los resultados obtenidos, trabajos relacionados, y también el planteo de trabajos futuros.

6.1. Análisis de Diseño y trabajos relacionados

Para la realización del presente trabajo se hizo un análisis previo de trabajos preexistentes relacionados con el área. Se encontró con que los documentos RFC[1] oficiales provistos por la IETF[2] que describen el sistema $DNSSEC$, no describen exhaustivamente el funcionamiento del mismo, dejando lugar a ambigüedades de interpretación, especialmente para su implementación. Se decidió seguir la línea de trabajo de [32], utilizando su formalización planteada para el sistema DNS como base, y a partir de la misma ampliarla para lograr modelar $DNSSEC$.

Al momento de modelar el sistema se consideró importante poder describir mayormente los aspectos relacionados a la cadena de confianza. Siguiendo este criterio, se detallan a continuación las decisiones de diseño tomadas durante la formalización en el cálculo de construcciones inductivas expuesta en el capítulo 4.

Los servidores $DNSSEC$ están preparados para interactuar con servidores DNS, salvo que al hacerlo pierden todas sus medidas de seguridad, por ejemplo si para las delegaciones se utilizan *glue records* en vez de registros de tipo *digest*, no se hace uso de ninguna medida de seguridad dejando obsoleta la cadena de confianza que arma $DNSSEC$. Por este motivo, al momento de formalizar el sistema se consideró uno en el cual no hay interacción entre protocolos, sino que el sistema de nombres está compuesto enteramente por servidores $DNSSEC$.

[1] Request for Comments
[2] Internet Engineering Task Force

En el modelado del sistema se hicieron ciertas suposiciones que simplifican la especificación y no afectan a los objetivos a analizar:

- En un mensaje $DNSMessage$ la sección *Question* puede contener más de una petición, mientras que en el modelo planteado se puede preguntar sólo por un Idx a la vez.

- Una llave ZSK (y KSK análogamente) puede ser un conjunto de registros, siendo la llave ZSK un $RRset$ en vez de un RR como se plantea en esta formalización.

- El sistema sólo permite realizar peticiones a servidores $confiables$ (e.g. ".") y de allí ir "acercándose" al servidor que contiene la respuesta buscada.

6.2. Conclusiones y resultados obtenidos

Al concluir este trabajo, se puede afirmar que los objetivos planteados en la introducción fueron logrados, se ha logrado una especificación formal de $DNSSEC$ que nos permitió modelar las propiedades que debe verificar un estado válido, y en base a esto, se plantearon ciertos invariantes que deben cumplirse para verificar que este sistema realmente soluciona los problemas de seguridad analizados para DNS.

La formalización en el CCI permitió realizar un análisis riguroso de las especificaciones propuesta por los RFCs más relevantes. En particular, se pusieron de manifiesto ambigüedades, imprecisiones y omisiones en los requerimientos funcionales de los comandos, se llevo a cabo un estudio de las alternativas planteadas como decisiones de implementación y se logró una formalización del sistema.

Contar con un modelo en el CCI permitió razonar de manera abstracta sobre su corrección. Por un lado se verificó que la semántica de los comandos conserva invariante la validez del estado, lo cual garantiza que el sistema no puede transicionar a un estado que no represente un posible estado DNSSEC. Por el otro se demostró que gracias a esta invariancia de la validez del estado, valen ciertas propiedades de seguridad fundamentales dentro del modelo, principalmente la validez de la cadena de confianza.

Gracias a la formalización del sistema se pudo encontrar una inconsistencia en los datos a través de la cadena de confianza, ya que al ejecutarse un rollover de la llave de zona (zsk), puede suceder que queden presentes datos en la vista de caché de un servidor que discrepen de aquellos verdaderos publicados en la vista autoritativa de otro. Para que esto no suceda, una posible solución sería que al momento de analizar si un $RRset$ debe ser descartado o no de una vista de caché, deba verificarse no sólo la expiración de su correspondiente TTL, sino también que este se mantenga válido y consistente a lo largo de toda la cadena de confianza.

6.3. Trabajos Futuros

La utilización del asistente de pruebas COQ permitió además obtener una especificación y posibilita, como trabajo futuro, la extracción de un prototipo correcto por construcción que sirva como implementación de referencia para desarrolladores e investigadores.

Si bien el desarrollo de esta tesina se basó particularmente en analizar la cadena de confianza construida por el sistema, existen otros aspectos introducidos en $DNSSEC$ que valen la pena discutir, por ejemplo el uso de los records de tipo $NSEC$ ($Next\ SECure$) presentados en el capítulo 3. Este tipo de registro provee mecanismos para dar respuestas autenticadas (firmadas por su correspondiente llave) al momento de denegar la existencia de un *RRset* solicitado. Para realizar este proceso, cuando no se encuentra un *RRset* buscado se responde con el registro $NSEC$ correspondiente al próximo *RRset* según el orden canónico de la vista en cuestión. Como consecuencia, un atacante podría realizar consultas de todas las entradas $NSEC$ de la zona y de esta manera aprender todos los nombres de dominio de la misma. Como trabajo futuro se propone un análisis riguroso de las consecuencias de seguridad que esta nueva característica trae, y el desarrollo de una extensión a la formalización en el CCI, que permita razonar sobre estos aspectos del sistema, formalizando el uso de NSEC para la prueba de no-existencia, e inclusive el uso de NSEC3 como alternativa.

Bibliografía

[1] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), Mar. 2005. Updated by RFC 6014.

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035 (Proposed Standard), Mar. 2005. Updated by RFCs 4470, 6014.

[3] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034 (Proposed Standard), Mar. 2005. Updated by RFCs 4470, 6014.

[4] T. Barros. Presentación OpenNIC 2008. Extensiones de Seguridad para DNS, November 2008.

[5] J. Bau and J. C. Mitchell. A security evaluation of dnssec with nsec3. In *NDSS*, 2010.

[6] S. Z. Béguelin. Especificación formal del modelo de seguridad de MIDP 2.0 en el Cálculo de Construcciones Inductivas. Master's thesis, Universidad Nacional de Rosario, May 2006.

[7] S. Bellovin. Report of the IAB Security Architecture Workshop. RFC 2316 (Informational), Apr. 1998.

[8] S. M. Bellovin. Security problems in the tcp/ip protocol suite. *SIGCOMM Comput. Commun. Rev.*, 19:32–48, April 1989.

[9] S. M. Bellovin and S. M. Bellovin. Using the domain name system for system breakins. In *Proceedings of the Fifth Usenix Unix Security Symposium*, 1995.

[10] Y. Bertot and P. Castéran. Interactive theorem proving and program development. coqárt: The calculus of inductive constructions, 2004.

[11] C. Cert/c. CERT Advisory CA-2001-02 Multiple Vulnerabilities in BIND, Aug. 2001.

[12] T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[13] D. Davidowicz. Domain Name System (DNS) Security, 1999.

[14] K. Davies. ICANN Presentation. DNS Cache Poisoning Vulnerability, November 2008.

[15] D. Eastlake 3rd. Secure Domain Name System Dynamic Update. RFC 2137 (Proposed Standard), Apr. 1997. Obsoleted by RFC 3007.

[16] D. Eastlake 3rd. Domain Name System Security Extensions. RFC 2535 (Proposed Standard), Mar. 1999. Obsoleted by RFCs 4033, 4034, 4035, updated by RFCs 2931, 3007, 3008, 3090, 3226, 3445, 3597, 3655, 3658, 3755, 3757, 3845.

[17] E. Gavron. RFC 1535: A security problem and proposed correction with widely deployed DNS software, Oct. 1993. Status: INFORMATIONAL.

[18] K. Harrenstien, M. Stahl, and E. Feinler. DoD Internet host table specification. RFC 952, Oct. 1985.

[19] K. Harrenstien, M. Stahl, and E. Feinler. Hostname Server. RFC 953 (Historic), Oct. 1985.

[20] M. Kennedy. Request for Comments Summary RFC Numbers 1700-1799. RFC 1799 (Informational), Jan. 1997.

[21] C. Liu and P. Albitz. *DNS and BIND*. Fifth edition, 2006.

[22] P. Martin-Löf and G. Mints, editors. *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*. Springer, 1990.

[23] The Coq development team. The Coq proof assistant reference manual, version 8.2, August 2009.

[24] A. Meulenkamp. Ripe NCC Presentation. Welcome to the DNSSEC RIPE NCC Training Course, 2005.

[25] P. Mockapetris. Domain names: Concepts and facilities. RFC 882, Nov. 1983. Obsoleted by RFCs 1034, 1035, updated by RFC 973.

[26] P. Mockapetris. Domain names: Implementation specification. RFC 883, Nov. 1983. Obsoleted by RFCs 1034, 1035, updated by RFC 973.

[27] 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.

[28] 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.

[29] P. U. D. of Computer Sciences and C. Schuba. *Addressing weaknesses in the domain name system protocol*. Number n.º 28 in CSD-TR / Computer Sciences Department, Purdue University. Purdue University, Dept. of Computer Sciences, 1994.

[30] J. Postel. Name server. IEN 116, Aug. 1979.

[31] C. D. Rosa. Especificación formal del modelo rbac en el cálculo de construcciones inductivas. Master's thesis, Universidad Nacional de Rosario, Argentina, Oct. 2008.

[32] K. N. L. Steven Cheung. A formal-specification based approach for protecting the domain name system. Technical report, University of California, Davis, 2000.

[33] Z. Su. Distributed system for Internet name service. RFC 830, Oct. 1982.

[34] Z. Su and J. Postel. Domain naming convention for Internet user applications. RFC 819, Aug. 1982.

[35] P. Vixie. Dns and bind security issues. In *Proceedings of the 5th conference on USENIX UNIX Security Symposium - Volume 5*, pages 19–19, Berkeley, CA, USA, 1995. USENIX Association.

[36] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. Dynamic Updates in the Domain Name System (DNS UPDATE). RFC 2136 (Proposed Standard), Apr. 1997. Updated by RFCs 3007, 4035, 4033, 4034.