Using a Byzantine-Fault-Tolerant Algorithm to Provide a Secure DNS

by

Zheng Yang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

Author ....
Department of Electrical Engineering and Computer Science
May 24, 1999

Certified by ....
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Committee on Graduate Students

Using a Byzantine-Fault-Tolerant Algorithm to Provide a Secure DNS

by

Zheng Yang

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

The Domain Name System, or DNS, is a distributed database that is used to provide a name service for the Internet. It has become a critical part of the Internet infrastructure. Because of its importance, DNS is a favorite target of malicious hackers. However, DNS is not designed to be a secure protocol. To make the DNS more robust, a DNS security extension has been proposed. In this extension, the authentication of the queried data can be verified by using a public-private key scheme. But this extension still has some security flaws.

This thesis analyzes the security issues of DNS and its security extension. It presents a design and implementation of a Byzantine-fault-tolerant DNS based on a new Byzantine-fault-tolerant algorithm. This DNS also support secure dynamic update operations. The malicious user needs to compromise at least $f+1$ replicas to effectively attack the system, which consists of $3f+1$ replicas. This thesis also shows that the Byzantine-fault-tolerant DNS performs almost as well as an implementation of the DNS security extension.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

I am greatly indebted to my advisor Prof. Barbara Liskov, for guidance, encouragement and support throughout the course of this study. Her insight, character and enthusiasm for research have profoundly influenced my own thinking and approach to academic research work. Her insightful comments have been most valuable for my thesis. The presentation of this thesis has been significantly improved because of her careful and critical proofreading.

Special thanks to Miguel Castro, whose great work on Byzantine-fault-tolerant algorithms triggered this project and let me run with it. Also, thanks for his invaluable advice and feedback. He patiently answered my questions, and proofread many extremely rough drafts. The replication library provided by him is really elegant and saved me a lot of time.

I would also like to thank Chandrasekhar Boyapati for his proofreading. I really enjoyed the technical and philosophical discussions with him, and the puzzles he gave to me. Thanks to ChonChon too, for his proofreading of this thesis.

My friends from the rest of our group made my stay at PMG a pleasant experience. I am grateful to Andrew, Arvind, Atul, ChonChon, Dorothy, Doug, Jason, Nick, Paul, Peter, Phil, Richard, Sameer, Shan-Ming and Xiaowei,

There are of course no words to describe my gratitude to my parents. They have always been there when I needed them. I only hope this work will make them proud of their son.

Without the love and support from my wife, Shuyi, I would never have achieved as much as I have today. She has sacrificed a great deal for my well-being. Simply saying thanks to her is far less than enough.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Problems

The Domain Name System, or DNS, is a distributed database that provides the name service for the Internet. It is a critical part of the Internet Infrastructure. All of the Internet's network services, which include the World Wide Web, electronic mail, remote terminal access, and file transfer, use DNS. Because of its importance, DNS is a favorite target of malicious attackers. The growing reliance of industry and governments on the information resources available on the Internet makes malicious attacks more attractive and makes the consequences of successful attacks more serious.

Meanwhile, DNS was not designed to be a secure protocol. DNS servers never authenticate the data that they send to DNS clients. Clients can only judge the origin and authenticity of the reply data they received by the senders' IP addresses included in the reply packets, which are easy to fake. In addition, DNS is vulnerable to a single node fault. A single compromised server can fool all clients that contact it and cause substantial damage.

To fix the security flaws in DNS, a DNS security extension has been proposed and partially implemented [EK97, Eas99a]. The main idea of the extension is to authenticate data in the DNS by pre-generating digital signatures for each data items of the DNS database. Servers return queried DNS data to clients along with the proper digital signatures. Clients use these signatures to verify the authenticity of the reply data. In a *zone transfer*, a process in which the whole database is transferred from one server to another server, a signature covering the whole database is used to verify the integrity of the database.

Unfortunately, this extension still has some security problems. For example, there is a dilemma about the way to store the private keys that are used to generate the signatures. To prevent a hacker from knowing the private key and being able to fake signatures when a server has been compromised, private keys are recommended to be kept and used in non-network connected, physically secure machines only. But this approach raises another issue. The off-line key cannot be used to generate signatures in real time to protect the dynamic update data in the database. Furthermore, the integrity of the database cannot be verified after any dynamic update of the database.

There are some other security issues in the DNS security extension. Generating a signature is very expensive, so signatures in the DNS security extension are generated only occasionally and have a quite long time-to-live (TTL). This opens a big window for freshness attacks, that is, stale data can be used to fool clients if its corresponding signature has not expired. In addition, the DNS security extension still uses a simple replica scheme to synchronize data between servers. This causes consistency and reliability problems. Clients may get stale data from some servers even without an intentional attack. Some updated data may be lost forever if a server that holds the new data crashes before transferring them to other servers.

It is clear that the DNS security extension is not secure enough, especially in the environment in which dynamic updates happen quite often. In fact, today's Internet is more dynamic than ever, and this trend will continue. Hundreds of thousands of hosts join the Internet everyday. Mobile computing, in which DNS data are highly dynamic, is also becoming more common. Furthermore, DNS is supposed to provide a global public key distribution infrastructure, but this must be a highly reliable service and always give the freshest authenticated data to clients.

In other words, we need a more secure DNS.

1.2 Our Contribution

To solve the above problems and build a more secure DNS, we designed and implemented a Byzantine-fault-tolerant DNS, or BFT-DNS. In the Byzantine fault model, faulty nodes may behave arbitrarily. So a Byzantine-fault-tolerant DNS is a hacker tolerant DNS.

There is a lot of work on agreement and replication techniques that tolerate Byzantine faults. However, most of this work either concerns techniques that were designed to demonstrate theoretical feasibility, which are too inefficient to be used in practice, or rely on known bounds on message delays and process speeds. However, the Internet is an asynchronized environment. The only technology that is suitable for our BFT-DNS is an algorithm recently invented by M. Castro and B. Liskov, which is described in [CL99a, CL99b].

Using M. Castro and B. Liskov's Byzantine-fault-tolerant algorithm, which we will refer to as CLBFT, our BFT-DNS provides both safety and liveness assuming that no more f replicas are faulty in a system consisting of $3f + 1$ servers. Safety means that the replicated service behaves like a centralized implementation that executes operations atomically one at a time. Liveness means that a client will eventually receive the correct reply from the servers.

In our BFT-DNS, each server has its own private key. A client sends a request to all the replicas of a server and waits for enough authenticated replies to accept a reply using the CLBFT algorithm.

We designed two different schemes for read requests in BFT-DNS. In the first scheme, session keys, which are established by using servers' public-private key pairs, are used to authenticate the communications between clients and servers. In the second scheme, we use pre-generated signatures to authenticate the replay data. In addition, a recently generated signature that includes the newest version number of the server's database with a very short time-to-live is returned to the client to indicate that the reply is *almost* up-to-date.

BFT-DNS supports dynamic update operations very well. An update operation is done independently in enough non-faulty replicas before replying to the clients to ensure system consistency. The dynamic updated data are protected as well as other data.

Our BFT-DNS is practical, yet it can provide highly secure service. Our performance test data shows that our BFT-DNS performs almost as well as the DNS security extension; in some cases it performs even faster.

Thus, this thesis makes the following contributions:

- It provides an analysis of the security flaws of current DNS and the DNS security extension.

- It describes the design and implementation of a Byzantine-fault-tolerant DNS, which is more secure than the DNS security extension.

- It provides experimental results that justify the practicability of the new Byzantine-fault-tolerant DNS.

1.3 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 gives a basic overview of the domain name system, that is, what is DNS and how it works. Chapter 3 describes the DNS security extension and analyzes some security issues of DNS and DNS security extension. Chapter 4 describes the design goal for BFT-DNS and how to design our BFT-DNS using the new Byzantine-fault-tolerant algorithm. Chapter 5 describes how we implemented our BFT-DNS based on an existing DNS software package and a Byzantine-fault-tolerant replication library. Chapter 6 describes the setup, configuration, and experiments used to study the performance of BFT-DNS. Chapter 6 also compares the performance of BFT-DNS with the DNS security extension and the DNS without the security features. Chapter 7 describes some possible extensions to BFT-DNS. Chapter 8 makes some concluding observations and describes some areas for future work.

Chapter 2

DNS

2.1 Overview

The Domain Name System, or DNS [Moc87a, Moc87b, Ste94, MD88], is a distributed database that is mainly used by applications to map between hostnames and IP addresses, and to provide other important information about the domain or the host, e.g., e-mail routing information. No single site on the Internet knows all the information in DNS. Instead, sites are organized in hierarchical domains. Each domain maintains its own database of information and runs a server program (*name server*) that any hosts across the Internet can query. An application accesses the DNS through a *resolver*. The resolver contacts one or more name servers to do the query. The resolver communicates with name servers using the TCP/IP protocols.

This chapter will begin with a description of the hierarchical DNS name space and its distributed administration structure, followed by the two types of active agents in DNS, the name server and the resolver. Then it will describe how the DNS queries are resolved. Finally, it will depict how the database in DNS is organized and what kind of information is in the database.

2.2 Name Space

The DNS name space is hierarchical. Figure 2-1 provides an illustration of the current DNS name space structure. Every node in the hierarchical tree has a label. The root of the tree is a special node with a null label. The domain name of any node in the tree is the list of labels, starting at that node, and walking up to the root, using a period to separate the labels. Every node in the tree must have a unique domain name, but the same label can appear at different points in the tree.

The top-level domains are divided into four areas:

1. arpa is a special domain used for address-to-name mappings, e.g., translating *arpa.in-addr.40.0.26.18* (ip address 18.26.0.40) to *chord.lcs.mit.edu.*

2. Seven 3-character domains are called the *generic domains*, or the *organizational domains.* See Figure 2-2.

3. The 2-character domains are based on the country codes found in ISO 3166, e.g., the .cn domain for China. These are called the *country domains.*

4. Seven proposed new domains, including *.arts*, *.firm*, *.info*, *.nom*, *.rec*, *.store*, and *.web*, are intended to accommodate the rapid expansion of the Internet and the need for more domain name space. These new domains are not fully functional yet.

Many countries form second-level domains beneath their 2-character country codes similar to the generic domains: *.edu.cn*, for examples, is for educational institutions in the China.

Figure 2-1 shows the hierarchical organization of the DNS, which is similar to the Unix filesystem. It also shows the basic idea of *zone*, which will be described in the next section.

2.3 Administration

The administration of the DNS is also hierarchical. The delegation of responsibility within the DNS is one of the most important features of the DNS. No single entity manages every label in the tree. Instead, one entity maintains a portion of the tree and delegates responsibility to other entities for specific zones.

A *zone* is a subtree of the DNS tree that is administered separately. A common zone is a second-level domain, *mit.edu*, for example. Many second-level domains then divide their zone into smaller zones. For example, MIT divides itself into zones based on departments or labs, e.g., *lcs.mit.edu* (for the Laboratory of Computer Science) and *media.mit.edu* (for the Media Lab).

Once the authority for a zone is delegated, it is up to the person who is responsible for the zone to provide at least two *name servers*, which contain the same zone information databases, for the zone. Whenever a new system is installed in a zone, the DNS administrator for the zone allocates a name and an IP address for the new

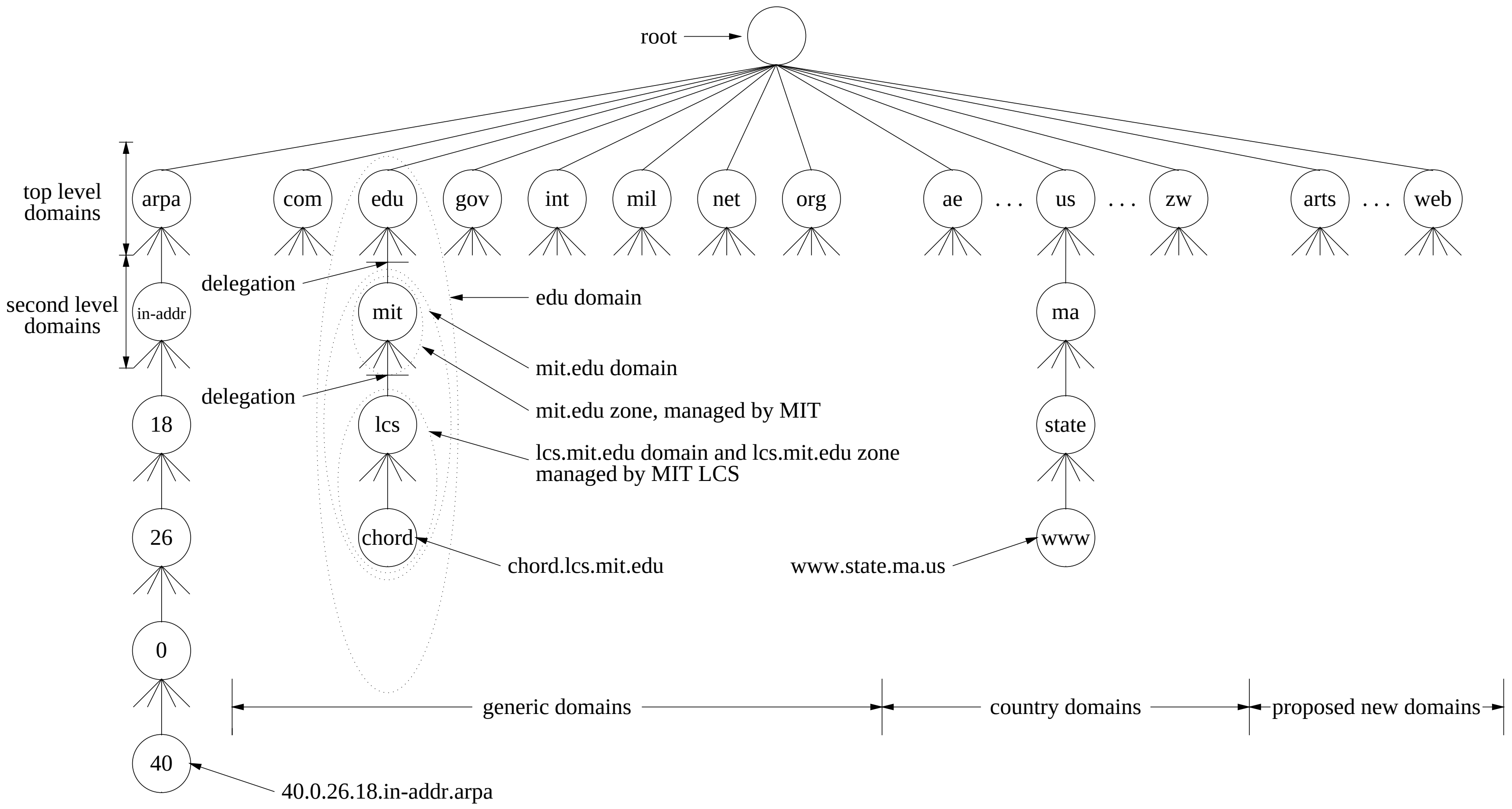


Figure 2-1: Hierarchical organization of the DNS

Domain	**Description**	**sample**
com	commercial organizations	oracle.com
edu	educational institutions	mit.edu
gov	other U.S. governmental organizations	whitehouse.org
int	international organizations	who.int
mil	U.S. military	navy.mil
net	networks	internic.net
org	other organzations.	un.org

Figure 2-2: The 3-character generic domains

system and enters these into the name server's database. The need for delegation is obvious. In places like a small startup company, a single person can manage the database and add new hosts to the database. But in a big corporation, which may have hundreds of thousands of computer systems, it is impossible for one person or even one group to keep up with the work. The issue is even more serious at the root domain. In this case, the responsibility would have to be delegated (probably by departments). Below are some sample zones with different sizes.

The Network Solutions Inc. [Int99], known as the internNIC (the Internet Network Information Center), maintains the DNS root and some top level zones. It is the exclusive registrar of names ending in .com, .net and .org. All in all, more than four million domain names are in its database, most of them are delegation points. Everyday there are six thousand new domain names are added to the database by the administrator of this zone according to users' requests from all over the world.

The *lcs.mit.edu* zone, maintained by the Computer Resource Service Group in MIT LCS, is the largest zone in the *mit.edu* domain. Its database contains 2885 domain names (April 6, 1999) and 4 delegation points. Most of the delegation points point to some highly dynamic subzones, e.g. *wireless.lcs.mit.edu.* The *lcs.mit.edu* zone is relatively stable.

Most of the zones in the format of *whatever.com* contain only 2 domain names, one for the web server and one for the mail server, and no further delegation point. They are usually owned by small businesses or individual people. They are very stable.

2.4 Name Servers and Resolvers

DNS has two types of active components: *name servers* and *resolvers*. Name servers are repositories of information, and answer queries using whatever information they possess. Resolvers interface to client programs, and embody the algorithms necessary to find a name server that has the information sought by the client.

2.4.1 Name Servers

A name server is said to have authority for one zone or multiple zones. The person responsible for a zone must provide a primary name server for that zone and one or more secondary name servers. The primary name server for a zone reads the data for the zone from a file on its host. A secondary name server for a zone gets the zone data from another name server that is authoritative for the zone, called its *master server*. In most case, the master server is the primary server, but it also can be another secondary server.

When a new host is added to a zone, or some records need to be changed, the administrator will modify the database in the primary server. Editing the database file directly and notifying the primary server to reload the data is the easiest and most common way to update the database. A secondary server queries its master server on a regular basis and if the master server contains new data, the secondary obtains the new data and updates its database. This is referred as a *zone transfer*. DNS uses this simple replica mode to improve availability. If one name server is down, resolvers can still use other name servers. Also workload can be distributed to get better performance. Section 2.6.2 provides more details on *zone transfer* .

2.4.2 Resolvers

Resolvers are the clients that access name servers. Programs running on a host that needs information about a domain name use the resolver. The resolver handles:

- querying name servers according to requests from programs
- interpreting responses, which may contain the requested data or an error
- returning the information to the programs that requested it

In most systems, the resolver is just a set of library routines that can be called from any program. The *gethostbyname* and the *gethostbyaddr* are the two most frequently called resolver functions.

2.5 Query Resolution

There are 2 kinds of queries: recursive and iterative.

Recursive queries place most of the burden of resolution on a single name server. After receiving a recursive query, the name server is then obliged to respond with the requested data or with an error stating that data of the requested type does not exist or that the domain name specified does not exist. The name server cannot just refer the querier to a different name server, because the query is recursive. Instead it must contact the *closest known* name servers. A recursive query can be sent to these *closest known* servers to oblige them to find the answer, but most name servers politely send iterative queries to other name servers "closer" to the domain name it's looking for.

In iterative resolution, a name server simply gives the best answer it already knows to the querier. The queried name server consults its local data and looks for the data requested. If it does not find the data there, it makes its best attempt to give the querier data that will help it continue the resolution process, generally, the IP addresses of the *closest known* name servers.

Closest known name servers are the servers authoritative for the known zone closest to the domain name being looked up. For example, for a query which looks up the address of the domain name *chord.lcs.mit.edu*, the name servers for zone *lcs.mit.edu* will be the *closest known* name servers if their IP addresses are known by the name server that received the query; the name servers for zone mit.edu will be the next candidates. If neither are known, the name server has to contact (or return the IP addresses of) the *root name servers.*

No name server knows how to contact every other name server. Instead every name server must know how to contact the *root name servers.* As of April 1999, there were thirteen root name servers distributed all over the world. The primary server *A.ROOT-SERVERS.NET.*, whose IP address is 198.41.0.4, is maintained by the Network Solution Inc and is located in Herdon, VA, US. All the name servers must know the IP addresses of each root server. The root servers then know the name and

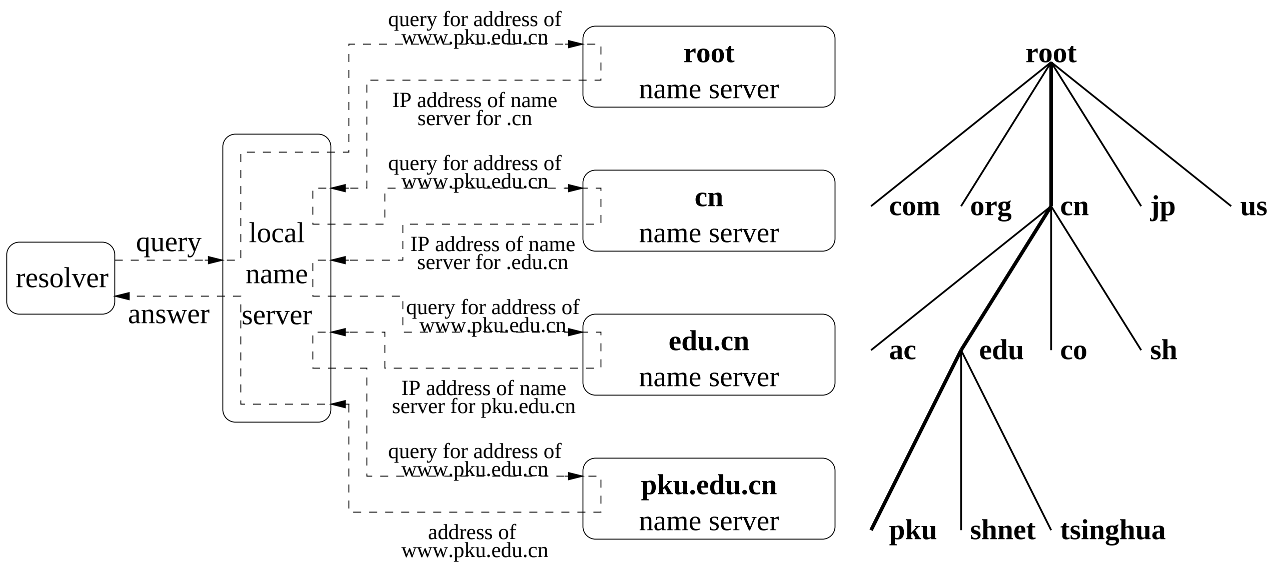

query for address of www.pku.edu.cn
root name server
IP address of name server for .cn
query for address of www.pku.edu.cn
cn name server
IP address of name server for .edu.cn
query for address of www.pku.edu.cn
edu.cn name server
IP address of name server for pku.edu.cn
query for address of www.pku.edu.cn
pku.edu.cn name server
address of www.pku.edu.cn
resolver
query
answer
local name server
root
com
org
cn
jp
us
ac
edu
co
sh
pku
shnet
tsinghua

Figure 2-3: A Resolution of a Recursive Query

IP address of each authoritative name server for all the second-level domains.

Caching is a fundamental property of the DNS. That is, when a name server receives information about a mapping, it may cache that information so that a later query for the same information can use the cached result and will not result in additional queries to other servers. This mechanism offloads the root servers and reduces the DNS traffic on the Internet.

2.5.1 Resolution Samples

Figure 2-3 shows the recursive resolution process for the address of a real host in a real domain, including the process corresponds to traversing the domain name space tree.

A resolver sends a query to a local name server for the IP address of *www.pku.-edu.cn*. The local name server searches its database and cache and finds no entry for *www.pku.edu.cn*, none for *pku.edu.cn*, *edu.cn*, or *cn*. So the local name server queries a root name server for the address of *www.pku.edu.cn* and is referred to the *cn* name servers. The local name server asks a *cn* name server the same question, and is referred to the *edu.cn* name servers. A *edu.cn* name server refers the local name server to the *pku.edu.cn* name servers. Finally, the local name server gets the answer from a *pku.edu.cn* name server and sends the answer back to the resolver. If the local name server happens to know the IP address of a name server of *edu.cn*,

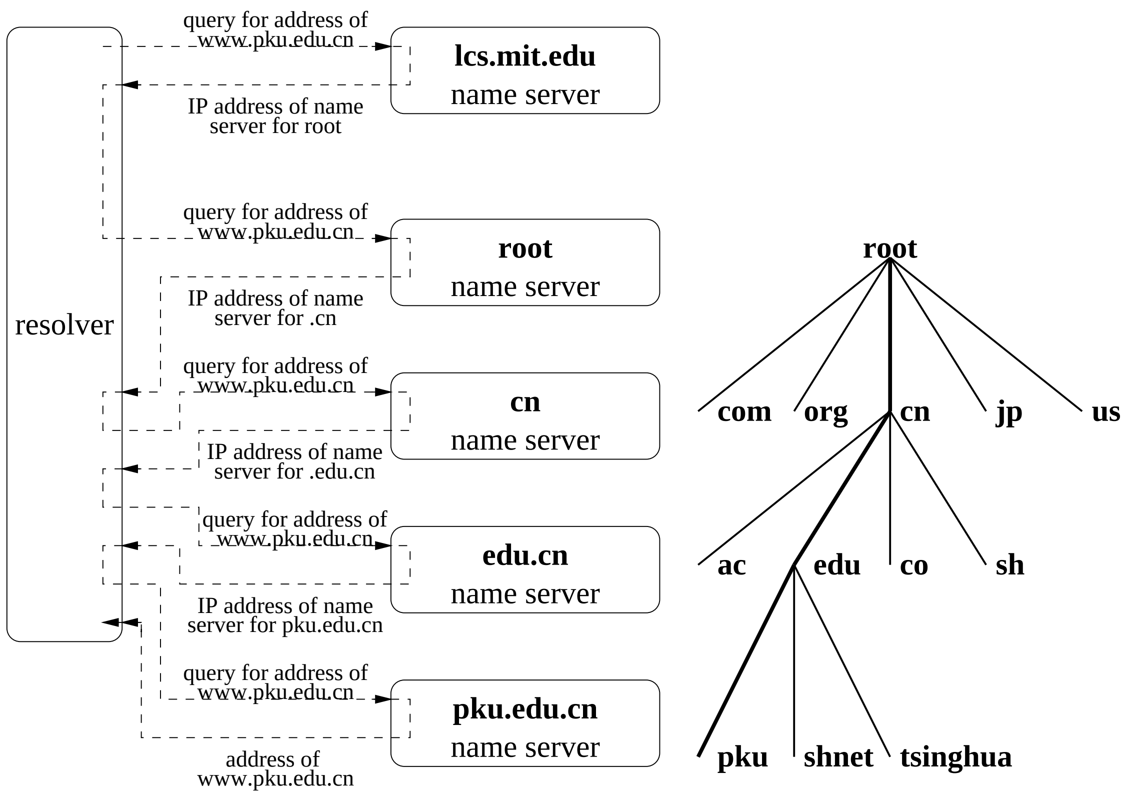

resolver
query for address of www.pku.edu.cn
lcs.mit.edu name server
IP address of name server for root
query for address of www.pku.edu.cn
root name server
IP address of name server for .cn
query for address of www.pku.edu.cn
cn name server
IP address of name server for .edu.cn
query for address of www.pku.edu.cn
edu.cn name server
IP address of name server for pku.edu.cn
query for address of www.pku.edu.cn
pku.edu.cn name server
address of www.pku.edu.cn
root
com
org
cn
jp
us
ac
edu
co
sh
pku
shnet
tsinghua

Figure 2-4: A Resolution of a Iteration Query

it can directly contact this name server without asking a root name server and a *cn* name server first.

Figure 2-4 shows the iterative resolution process for the address of a real host in a real domain. This process is similar to the process in Figure 2-3, except that the resolver is smarter here. The process corresponds to traversing the domain name space tree is handled by the resolver itself, instead of the local name server.

2.6 Resource Records

The database in a name server consists of data entries called DNS *resource records*, or RRs. A resolver sends a request to a name server to ask for one RR or a set of RRs.

2.6.1 Format

All of the DNS resource records have a similar format (Figure 2-5).

The first field in any RR is always a domain name that is in the domain name

domain name	**TTL** (optional)	**class**	**type**	**data**

Figure 2-5: The Format of DNS RRs

space. It is also called *owner* of the RR.

The second optional field is a Time-to-Live (TTL) value, which indicates the length of time that the information in this RR should be considered valid. It is the amount of time that any name server is allowed to cache the data. After the TTL expires, the name server must discard the cached data and optionally get new data from the authoritative name servers; otherwise, changes to that data on the authoritative name servers would never reach the name servers that cache the old data.

The third field indicates the RR *class*. Although in today's DNS databases the string 'IN' is most likely to be found to indicate that the RR is for Internet, this field is still present for historical purposes and compatibility with older systems.

The fourth field indicates the *type* of RR. This field is followed by a data field that is specific to the type. There are about 40 different types, some of which are now obsolete. Figure 2-6 shows some common and important types and we will describe them in detail in the following subsection.

type	**description**
SOA	start of authority
A	IP address
NS	name server
CNAME	canonical name
PTR	pointer record
MX	mail exchange record

Figure 2-6: Some important RR types

The combination of $< domainname, class, type >$ is not a primary key for RRs, i.e., there can be multiple RRs with the same name, class and type in one zone.

2.6.2 Start of Authority Record

Each zone has one and only one start of authority (SOA) record. It contains some important information about the zone. The following SOA record example comes from the *lcs.mit.edu* zone.

```
lcs.mit.edu.  180 IN SOA mintaka.lcs.mit.edu.  bug-lcs-domain.lcs.mit.edu.(
                              252879555 ; serial number
                              900 ; refresh (15 mins)
                              300 ; retry (5 mins)
                              259200 ; expire (3 days)
                              1800 ) ; TTL (30 mins)
```

The SOA record includes the name of the primary name server for this zone and the e-mail address of the person responsible for the name server. Note that the e-mail address is not in the expected form; rather than *user@domain*, the format *user.domain* is used. So in the above example, *bug-lcs-domain@lcs.mit.edu* should be used to report a bug.

The SOA record also contains five other parameters in its data field.

1. The *serial number* identifies the version of the DNS database. Whenever the file's information is changed, the serial number must be incremented so that secondary servers can detect changes and keep their database copies up-to-date when they examine the SOA record in the primary server. The simplest way is just to start counting at 1 and increment it every time there's a change.

2. The *refresh* value tells the secondary name servers how often to check for updated information.

3. The *retry* value tells the secondary name servers how often they should reattempt a connection in case that they are unable to contact the primary name server.

4. If a secondary name server cannot contact the primary name server for *expire* seconds, the secondary will stop answering any queries about this domain.

5. The TTL value is the default TTL value for all RRs in this zone. It may be overridden by a TTL value provided with a given RR.

2.6.3 IP Address Record, Pointer Record and Canonical Name Record

The A RRs, which are the most common RRs, contain an IP address to be associated with the domain name in the first field of the record. One domain name can be mapped to several different IP addresses. Some heavily loaded web sites use this trick to distribute the incoming HTTP requests to different servers. Also different domain names can be mapped to one IP address. In this way a single machine can host lots of small web sites (what is called virtual host).

Example: (from the zone *lcs.mit.edu*):

```
chord.lcs.mit.edu.  1800 IN A 18.26.0.40
```

The PTR RRs do the reverse of what as the A RRs do. They are used to map an IP address to a human-understandable domain name. The IP address is represented as a domain name in the in-addr.arpa domain.

Example: 40.0.26.18.in-addr.arpa. 1800 IN PTR chord.lcs.mit.edu.

The A CNAME (stands for canonical name) RRs contain an alias domain name to associate with the domain name in the first field of the RRs.

Example (from the zone *lcs.mit.edu*):

```
zheng.lcs.mit.edu.  1800 IN CNAME dialup-35.lcs.mit.edu.
```

2.6.4 Name Server Records

Name server records, or NS RRs, specify the authoritative name servers for a domain. They are represented as domain names in the data fields. Usually when NS RRs are requested, the corresponding A RRs containing the IP addresses of the name servers will also be sent to the resolver. So a second query for the IP addresses of name servers is avoided. There are at least two NS RRs associated with one domain name. The NS RRs are the delegation points for the parent-zone and its child-zones (See Section 2.3). They direct a resolver or a name server to send its request to the name servers that maintain related information and finally get the answer.

Example: (from the zone *mit.edu*):

```
lcs.mit.edu.  21170 IN NS MINTAKA.lcs.mit.edu.
lcs.mit.edu.  21170 IN NS OSSIPEE.lcs.mit.edu.
```

2.6.5 Mail Exchange Records

Mail exchange records, or MX RRs, contain the address of the mail exchange system(s) for this domain. The number before the address is the *preference* of this mail exchange system.

Example: (from the zone *mit.edu*):

```
mit.edu.  21600 IN MX 10 SOUTH-STATION-ANNEX.mit.edu.
mit.edu.  21600 IN MX 100 PACIFIC-CARRIER-ANNEX.mit.edu.
```

When a remote user sends mail to *zyang@mit.edu*, the remote mail system looks up the MX record for the mit.edu domain. The remote mailer will then attempt to establish a Simple Mail Transfer Protocol (SMTP) connection with the mail system with the lower preference value (in this case, *SOUTH-STATION-ANNEX.mit.edu.*). If that system is not available, the remote mailer tries the next level mailer, in this case, *PACIFIC-CARRIER-ANNEX.mit.edu.*.

Chapter 3

DNS Security Issues

3.1 Overview

DNS was not designed to be a secure protocol. The protocol contains no means to verify whether the information returned by a DNS query is correct. Meanwhile, DNS plays a key role in directing the information flow in the Internet. Furthermore, the RRs queried from DNS sometimes are even used for checking authentication in other Internet protocols. As a result, DNS is a favorite target of hackers. The security weakness of DNS has received more attention recently and a new security extension has been proposed.

This chapter starts by describing several common attacks to DNS. Then it describes the proposed secure DNS extension and explains why it is not sufficient.

3.2 Attacks on DNS

A resolver has no means to check the authenticity and integrity of the data returned by the name servers. So it has to trust the answer it got. Malicious users can take advantage of this security hole and the consequences can be serious.

3.2.1 How to Attack

Generally speaking, there are 3 ways to attack the DNS: name server break-in, data packet spoofing, and sending a fraudulent update request.

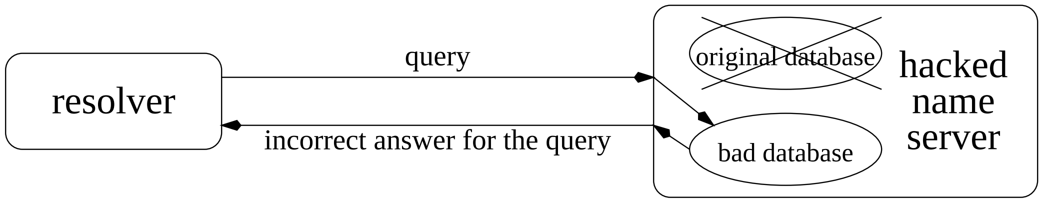


Figure 3-1: Break into a Name Server to Attack DNS

Break into the Name Server

The most straightforward way to attack DNS is to break into the name servers. Figure 3-1 provides an illustration. Although a name server is normally running as a process belonging to *root*, and the name server's files are all owned by *root*, it is still possible for an intruder to break into the system and gain *root* privileges, especially in some old operating systems that have well-known bugs and are not properly patched. Then the intruder can do whatever he/she wants to. The database files can simply be modified as the intruder wishes.

It is the best case for an intruder if he/she can break into the primary server of a zone. The secondary servers will contact the primary to ask for new data periodically according to the *refresh* value in the zone SOA RR. They cannot tell whether the data in the primary are valid and have to trust every byte the primary gives them. So all the name servers in this zone will eventually contain the bad database set by the intruder.

Spoof Network Packets

In most cases, the network data packets containing the answer to a query will pass some routers or gateways in the Internet before they reach the resolver that sent the query. If one of these routers or gateways is controlled by a malicious user, he/she can easily replace the packets containing the correct answer with packets containing the answer that he/she intends to give to the resolver. This is shown in Figure 3-2 .

Figure 3-3 shows another type of spoofing attack. The attacker no longer needs to control any name servers or routers, he/she just needs to sit in the way that the query packet will pass, sniff the packet, then generate and return a faked answer packet to the resolver *fast enough*. The source IP and source PORT fields in the packet's IP header should be set properly so that the resolver will think the packet is from the name server that it just sent the request to. *Fast enough* means the faked answer

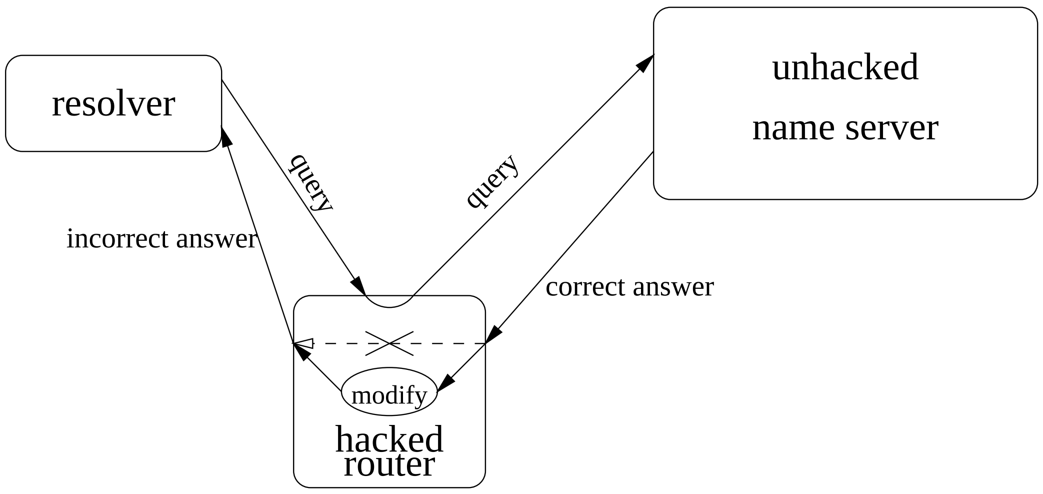


Figure 3-2: Break into a Router to Attack DNS

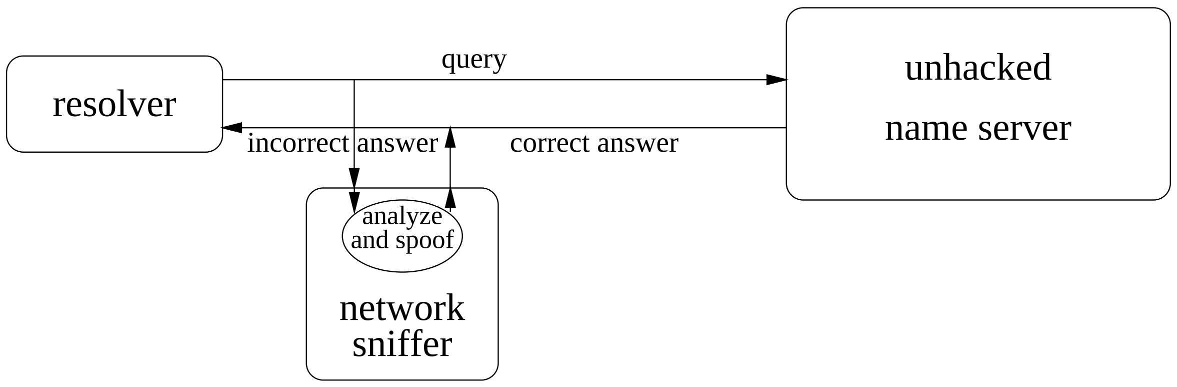


Figure 3-3: Sniff and Spoof to Attack DNS

packet must arrive in the resolver earlier than the correct answer packet from the name server. Since the resolver always takes the first answer, which is a fake one in this case, the correct one will be ignored. This type of attack is easy to mount if the attacker is in the same local network as the target resolver or name servers, since it is easy to do packet sniffing and spoofing in the Ethernet environment. The attack can also be used to transfer an invalid data set to a secondary name server when a zone transfer happens.

Send a Bogus Update Request

Although it seems naive, sending a fraudulent update request to the zone administrator to change some RRs with harmful values is very effective sometimes. The

DNS specification says almost nothing about the update. Normally a zone administrator takes care of all the RRs in the primary server and edits the database files manually when necessary. If the zone is large, it is impossible for him/her to check the validity of the RRs himself/herself. So he/she has to depend on the collaboration of users who should take responsibility for the validity of individual RRs in the database. For example, the *root* user of host *chord.lcs.mit.edu* should notify the administrator of *lcs.mit.edu* zone to update the database after changing the IP address of *chord.lcs.mit.edu*. The problem is that it may be difficult to check the authenticity and authority of the update request. In local cases, where users can know each other, this kind of attack may still be only one call away, if the attacker can mimic some authorized users perfectly. Protecting the RRs from unauthorized change is a very difficult problem, especially in some large domains, in which the administrator and the users only contact each other electronically.

3.2.2 The Consequence of a Successful Attack

It is clear that DNS is easy to attack. What kind of damage can this attack do? By giving incorrect direction information, a malicious hacker can do different bad things in different circumstances. Here are some simple examples.

Block Information

The hacker can shut down the name servers of *mit.edu*. Then users all over the world can no longer use the human understandable domain names to access any hosts in the mit.edu domain. Since few people can remember the IP address of *web.mit.edu*, the information that MIT wants to give to the public is blocked.

Give Wrong Information

When a resolver asks for the IP address of *www.stockquote.com*, the hacker can return an IP address of a web site built by himself/herself. Then the user may be fooled by the information from this bad site and as a result, make a wrong decision. This case is more dangerous, for wrong information is worse than no information most time.

Steal Valued Information

This case is similar to the former one, but even more dangerous. For example, when a resolver asks for the IP address of *homelink.bankboston.com*, which enables user do transactions on line, the hacker can return an IP address of a web site build by himself/herself. This web site would look just like the authentic one, so the user would input his/her user ID and password without any doubts. The hacker can then login into the authentic site immediately and get the user's personal account information to continue fooling the user. The user will never notice that the password is stolen before the next bank statement arrives.

Re-direct Information

By simply changing the MX RRs for a domain, a hacker can redirect all the incoming e-mails for a victim domain to his/her own machine. Then he/she can forward some of them to the authentic mail server while selectively keeping some.

Break into a System

Some protocols, for example, some old versions of Berkeley UNIX *r* commands (*rsh* and *rlogin*), use the hostname for authentication. A user can *rlogin* to a host from a trusted host without typing the password, given his/her usernames and UIDs are the same in two hosts. Some operating systems just examine the IP address of an incoming TCP connection, query the hostname of this IP address, and trust that the resulting hostname is correct, which it may not be. Some more robust systems perform a double check, that is, query the IP address for the hostname to see if the looked-up IP address matches the IP address of the incoming TCP connection. If a hacker can sniff and spoof in the local network to attack DNS at the right time, this double check would be in vain. Using IP addresses instead of hostnames for authentication can partly fix this problem, but is less acceptable to the users. Ordinary people can remember a hostname easily but not a meaningless 32-bit IP address.

Actually, today's TCP/IP provides no way to authenticate the origin of an IP packet. Therefore using hostname, or even IP address, for authentication is not safe, It should be used only as an auxiliary authentication method, or in cases where there is no other proper solution.

3.3 Proposed Secure DNS Extension

To make the DNS infrastructure more robust, a security extension to DNS was proposed in late 1997 [EK97, Eas99a]. It can provide data integrity and authentication to security-aware resolvers and applications through the use of cryptographic digital signatures.

3.3.1 Basic Idea

In the DNS security extension, authentication is provided by associating cryptographically generated digital signatures with RRs in the DNS. Commonly, there will be a single private key that authenticates an entire zone but there might be multiple keys for different cryptograph algorithms. If a security-aware resolver reliably learns a public key of the zone, it can authenticate the signed data read from that zone.

3.3.2 New Resource Record Types

Some new RRs *types* are defined to accomplish the authentication procedure in the DNS security extension. Below we describe some important new RRs. More detailed information can be found in [Eas99b, Eas99c].

Key Resource Record

The KEY RR is used to store a public key that is associated with a domain name. This can be the public key of a zone, a host, a user, or other end entity.

To construct a trusted authentication chain, which will be described below, a secure zone must contain KEY RRs for every delegated subzone. These KEY RRs also need to be signed by the superzone private key.

In addition to sending KEY RRs to resolvers when receiving an explicit request, a security-aware DNS server should include KEY RRs as additional information in its responses in the following cases:

1. On the retrieval of NS RRs, the KEY RRs with the same name should be included as additional information together with the corresponding A RRs. Here the KEY RRs always contain public keys for the queried zone.

2. On the retrieval of Type A RRs or AAAA RRs(for ipv6, similar to A RRs), the KEY RRs with the same name should be included. In this case, the KEY RRs

contain public keys for the queried host.

A KEY RR can store more than just DNS keys. Any application or protocol can store public keys in DNS. Each KEY is associated with a category - it is either a Zone key (associated with a zone), a Host/End Entity key (associated with a host), or a User key (associated with a user: DNS can store user names). Additionally, each KEY RR refers to a protocol that it is used with, such as DNS, email, etc. So we can use DNS as a Public Key Infrastructure.

Signature Resource Record

The SIG or "signature" RR is the fundamental way that data are authenticated in the secure DNS.

A SIG RR unforgably authenticates the set of RRs with a particular type, class, and name and binds it to a time interval and the signer's domain name. This is done by using the private key that belongs to the owner of the zone (called *signer*) from which the RRs originated to sign the digested contents of RRs plus other related information. Except for the SIG RRs, each type of RR has its corresponding SIG RR.

For every authenticated RRs set the query returns, the security-aware DNS server should send the available SIG RRs that authenticate the requested RRs set.

Next Resource Record

The SIG RR mechanism provides strong authentication of RRs that exist in a zone. But without the NXT (stand for *next*) RR mechanism, it is impossible to verify the existence of a name in a zone or a type for an existing name when a negative answer is received.

The nonexistence of a name in a zone is indicated by the NXT RR for a name interval containing the nonexistent name. A NXT RR and its SIG RR will be returned by a secure DNS server if it finds out that a requested RR is nonexistent.

The owner name of the NXT RR is an existing name in the zone. Its DATA field contains a "next" name and a type bit map. Thus the NXT RRs in a zone create a chain of all the literal owner names in that zone. The presence of the NXT RR means that no name, according to lexicographic ordering, between its owner and the name in its DATA field exists, and that no other type exists under its owner name. For example:

```
fix.foo.bar.  300 IN NXT frodo.foo.bar.  NS SIG KEY NXT
```

means there is no domain name existing between *fix.foo.bar* and *frodo.foo.bar*, for example, *fox.foo.bar* does not exist. Also *fix.foo.bar* only has NS RR, SIG RR, KEY RR, and NXT RR under its name.

Transfer Zone of Authority Record

Transfer Zone of Authority Record, or AXFR RR, is a special type of SIG RR. The primary name server generates this RR by computing a digest of all the RRs, including all SIG RRs except the AXFR RR, then signing it using the zone private key. This RR is used to prove the integrity of the RRs database when a zone transfer happens, i.e. a secondary wants a whole copy of the database.

3.3.3 Authentication Chain

A resolver could learn a public key of a zone either by having it statically configured within it or by querying it from the DNS, just like querying other types of RRs. In some cases, KEY RRs will also be returned as additional information without explicit query.

To reliably learn a public key by querying it from the DNS, the key itself must be signed with a key the resolver trusts. The resolver must be configured with at least a public key that authenticates one zone as a starting point. From there, it can securely read public keys of other zones. Generally, the resolver will be able to get the KEY RR for the root zone after the first round of the query. It can then descend within the tree of zones until it gets the final answer.

Figure 3-4 shows the idea of a trusted authentication chain in a sample resolution in secure DNS.

3.3.4 Dynamic Update

The secure DNS extension defines a new DNS request opcode, new DNS request and response packet format [VTRB97, Eas97]. An update request can specify complex combinations of deletions and insertions of RRs. Updates occur at the primary server of a zone. A request SIG must appears at the end of an update request and authenticate the request with the key of the submitting entity. It is the primary means of authenticating the request. The authority policy is determined by the zone administrator. A reasonable policy might be:

1. An update request signed by a zone private key has the authority to update any RRs in the zone.

2. An update request signed by a private key belonging to a domain name has the authority to update any RRs owned by this domain name. For example, the user who possess the private key of chord.lcs.mit.edu should be able to convince lcs.mit.edu zone to update the RR that contains the IP address of chord.lcs.mit.edu, by signing the update request using the private key.

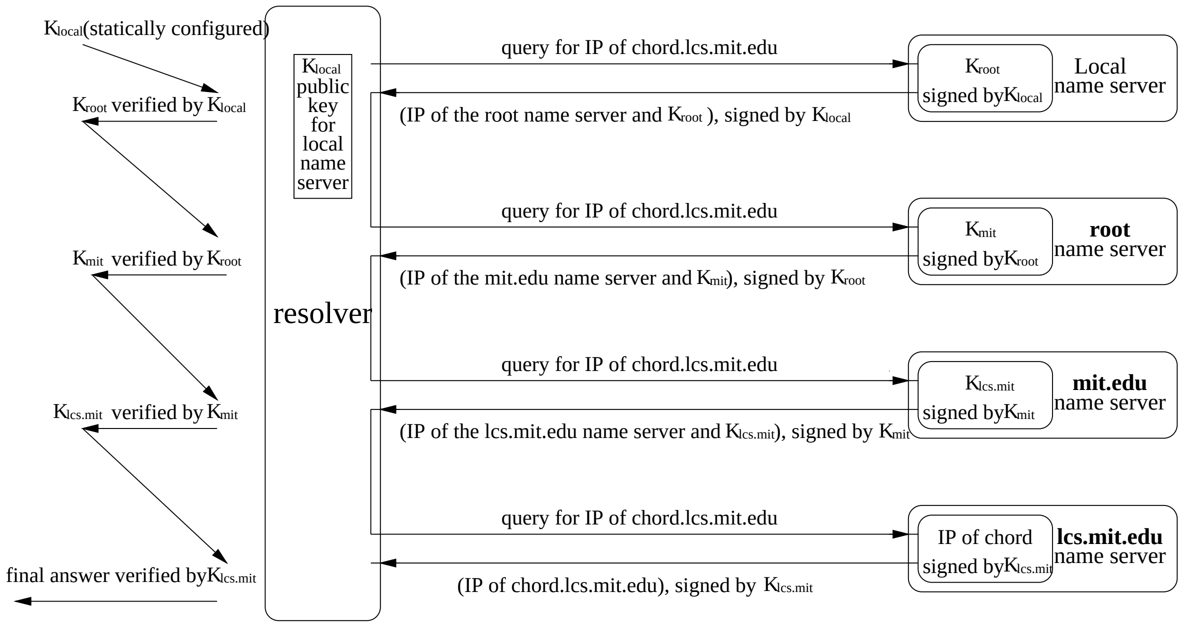
A *dynamic secure* zone is any secure DNS zone that can interpret a update request, check the authentication and authority of the request, and update the RR database in real time correctly. To check the authenticity of incoming update requests, it must contains some entity or user KEY RRs that can authorize dynamic updates. To update the RRs database correctly, the primary server must update all related RRs, for example, SOA RR, SOA SIG RR, NXT RRs, NXT SIG RRs, besides the requested one. Also a new AXFR RR should be computed, or the primary will not be able to convince a secondary in the next zone transfer.

Based on different security trade-off strategies, there are two basic kinds of dynamic secure zones, storing the zone key online and storing the zone key off-line. If the private key is online, the new SIGs related to the updated RRs can be computed immediately. If the private key is offline, that is, not accessible when update occurs, there will be some security problems, which will be discussed in the next section.

3.4 Attacks to Secure DNS Extension

The secure DNS extension makes verifying the authentication of answers possible. Therefore it patches the biggest security hole of DNS.

Attacking the secure DNS extension is harder than before, but not impossible. Depending on how the private key of a zone is stored, there are different ways to attack.



K_local (statically configured)
K_root verified by K_local
K_mit verified by K_root
K_lcs.mit verified by K_mit
final answer verified by K_lcs.mit
K_local public key for local name server
resolver
query for IP of chord.lcs.mit.edu
(IP of the root name server and K_root), signed by K_local
(IP of the mit.edu name server and K_mit), signed by K_root
(IP of the lcs.mit.edu name server and K_lcs.mit), signed by K_mit
(IP of chord.lcs.mit.edu), signed by K_lcs.mit
K_root signed by K_local
Local name server
K_mit signed by K_root
root name server
K_lcs.mit signed by K_mit
mit.edu name server
IP of chord signed by K_lcs.mit
lcs.mit.edu name server

Figure 3-4: A sample Resolution in secure DNS.

3.4.1 Zone Key is Online

Keeping the zone key in the disk or memory of the primary name server for a zone is the easiest way to store the key. In this mode, when authenticated updates succeed, new SIGs for the updated RRs can be calculated immediately. But this mode has a big disadvantage. The ultimate zone authority private key is stored in the primary name server, which is accessible to a remote attacker. This means that if the primary server is compromised, the intruder can steal this private key and sign whatever he/she wants. This attack is not essentially different from the attack in shown Figure 3-1. False data could be authenticated to secondaries and other servers/resolvers. Thus this mode cannot survive a single node failure, if the primary is the faulty node.

Notice that storing the encrypted private key in the server and decrypting it before using it to generate SIG RRs does not solve the problem. To be fully secure, the private key should never exist in the memory of the server, even for the shortest period. The intruder may dump and analyze the memory to grab the private key, or he/she can easily deploy a Trojan horse to capture the password that will release the ultimate power.

Smartcards, which are easy to attach or detach from a computer, can store the private key and do the signing without leaking the private key. This seems to be a good choice, except that this would need special hardware. But actually, this is not perfect either. Once the smartcard is attached to a compromised server, the intruder can use the stolen password to ask the smartcard to sign some bad RRs, although he/she cannot know the private key. Furthermore, stealing a private key from a smartcard is not impossible [AK96].

3.4.2 Zone Key is Off-line

Knowing the danger of storing zone private keys online, [Eas97] recommends that, where possible, zone private keys should be kept and used in off-line, non-network-connected, physically secure machines only.

Keeping a zone private key off-line while keeping a dynamic update key, which is authorized by the private key, online is not different from keeping the zone key on line. The hacker can break into the host that holds the online dynamic update key and generate the desired "dynamic update" RRs. Although the dynamic update key and the dynamic updated RRs can be revoked as soon as the zone administrator notices the invasion, the intruder may have done serious damage.

Implementing a truly off-line key storage mode is not trivial. One safe solution is to keep the host that holds the private key in a guarded and network isolated room, and to use removable media, such as a diskette, to transfer the update request and the signed new data between the signer (key holder) and the primary name server.

However, it is difficult for dynamically added or changed RRs and the related NXT RRs to be signed by the off-line zone private key. As a consequence, they are either not available or not protected by the zone private key before the next signing, which generally does not happen frequently. In some circumstances, the update request is time critical. Suppose a user is suspicious that one of his/her private keys is compromised; he/she will surely want the corresponding public KEY RR and SIG RR to be revoked and replaced immediately. But if the private key of the zone is stored off line, the user may have to wait until the next signing point to get relief. Furthermore, since the integrity of the whole RR database is not protected by the SOA RR, NXT RRs, and the AXFR RR, the secondary servers will not be able to know whether the RRs data the primary gives to them is complete and correct. This will introduce some attractive chances to hackers.

3.4.3 Freshness Attack

Both the online and offline modes are vulnerable to a freshness attack. A freshness attack occurs when a message (or message component) for a previous run of a protocol is recorded by an intruder and replayed as a message component in the current run of the protocol.

Since in DNS secure extension, the SIG RRs are not generated per-query but are pre-computed, they are vulnerable to a freshness attack.

A SIG RR is valid from the “signature inception” time until the “signature expiration” time. Also an “original TTL” field is included and protected by the signature. The corresponding RR(s) are valid until the “signature expiration” time or the TTL expiration time, whichever comes first. If an RR is updated before it expires, a resolver might still think that the old RR is the correct one. For example, suppose a KEY RR stores a public key belonging to a user U, and the corresponding SIG is generated at time T_s and will expire at time T_e. At time T_u, U suspects that his/her private key is compromised and sends an update request to the zone administrator to update the KEY RR and SIG RR. Suppose the update is successful. U is right, a hacker H stole U's private key at time T_h. He also keeps the old KEY RR and SIG RR. Then H can use the method shown in Figure 3-2 or Figure 3-3 to replay the old KEY RR and SIG RR when a resolver asks for U's public key. This can fool the resolver and make U's update ineffective until T_e. H can enjoy the stolen private key from T_h to T_u, even though U may notice and try to stop it.

3.4.4 Other Problems

The DNS security extension still uses the simple replica scheme for zone transfers. The relation between a primary name server and secondary name servers is very loose. Only the primary server is involved in an RRs update transaction. The secondary name servers know nothing about the update until the next zone transfer. Before that, the secondary name servers may return stale RRs. Furthermore, if the primary name server crashes and loses its state, the newly updated RRs will be lost forever, since the secondary name servers do not have this information yet. Attackers can take advantage of this protocol weakness to mount some attacks.

Chapter 4

Designing a Byzantine Fault Tolerant DNS

4.1 Overview

In this chapter, we will first analyze what the intrinsic security problems of the current DNS are. Next we will describe the design goal of the new DNS protocol we propose. After that, a recently-invented practical Byzantine Fault Tolerant algorithm will be introduced, followed by a new DNS system model and some detail read and update schemes. Finally we will discuss ways of attacking to the new DNS.

4.2 Intrinsic Problems with the Current DNS

As we have discussed in the last chapter, the current DNS provides no security, and the DNS security extension is not secure enough, especially for dynamic updates. This is because DNS has some intrinsic problems. The original designers of DNS simply did not expect the DNS to be the basis of the Internet; nor did they expect the Internet to evolve so fast, so they did not take security issues into serious consideration when they designed the protocol. The security extension also inherits some problems from its ancestor. Below are some fundamental reasons that cause the current DNS, even with the security extension, to not be secure enough.

4.2.1 Lack of Authentication

In the current DNS, there is no authentication method. It is clearly the biggest security hole of DNS. We should not blame the original DNS designers. They had good reason at the time DNS was proposed(12 years ago):

1. Computers were too slow to use public-private key encryption algorithms (PPK). Today's computers are at least 100 times faster, but using PPK still take a considerable amount of time.

2. No royalty-free practical PPK algorithms were available; even today most good PPKs collect royalty. The patent on the most popular RSA PPK algorithm will expire in September 2000.

3. The absence of serious and malicious hackers. At that time, the Internet was only used for research purposes and was only available in some big universities and big computer companies. In contrast, today the Internet is the biggest potential commercial platform and is available everywhere.

Secure DNS patches the hole, but not completely. There is no per-query-based authentication, so the resolver can still be fooled by replaying stale data.

4.2.2 No Consistency Control

The current DNS was evolved from the old *HOST.TXT* file approach. In the early days, the DNS service was provided by putting information about every host in the Internet into a simple file, HOSTS.TXT, which was centrally maintained on a host at the SRI Network Information Center (SRI-NIC) and distributed to all hosts in the Internet via direct and indirect file transfer. Although DNS is administrated hierarchically now, the essential way to propagate the new data has not changed. There is still no consistency control among the primary and secondaries. The states of the primary server will diverge from those of the secondary servers after an update and before the next zone transfer. As we mentioned in the last chapter, secondary servers will almost always contain some stale data, especially in highly dynamic zones.

4.2.3 Centralized Service, No Single Node Fault Tolerant

The problem has existed since the birth of DNS, and is not totally solved by the DNS security extension. Although there are several name servers in one zone, a resolver treats them separately. It will contact a second server only if the first one is down or the request times out. So one compromised server is enough to cheat the resolver. In the DNS security extension, things are better. A resolver will accept a replayed information in the worst case. But the updating process in a secure zone is still centralized in the primary. So the result of a primary server fault may be disastrous, either leaking the private key, therefore compromising the whole zone, or losing the new updates that have not been transferred to the secondaries.

4.3 Design Goals

To solve all of the problems mentioned above and to make DNS a more secure system, a newly designed DNS should have the following properties.

Consistency

The new system must ensure that the data in all non-faulty name servers of a zone are consistent. Thus, whenever an update request is received by a server, this request should be propagated to all other servers. More importantly, non-faulty servers should take the same action in response to the update request, that is, all nodes execute the same update operations in the same order, if the request is valid and authorized, or all nodes reject the request. Because of the existence of faulty servers, some non-faulty servers may contain stale data temporarily. The bottom line is that a resolver should not be confused by different answers and should always be able to get the newest data.

Reliability and Availability

The new system must provide service with high reliability and availability. Reliability is attained by ensuring that the information in the system is not lost or corrupted when some nodes fail. Reliability also guarantees that the effects of an operation executed on some data are visible to all those that access the data after the operation has completed regardless of any failure that might take place. Availability is attained by ensuring that, in addition to reliability, the data are always accessible when needed, even in the presence of some node failures.

Byzantine-Fault-Tolerance

The new system must be able to tolerate *Byzantine failures* [LSP82]. Most fault tolerant systems generally can only tolerant *fail-stop failures* [Sch90], which are intended to model unpredictable processor crashes. Fail-stop fault tolerance is not strong enough for the DNS because malicious attacks are becoming increasingly common. This kind of attack can cause faulty nodes to exhibit arbitrary behaviors, which are far beyond the fail-stop failure model. The Byzantine failure model is intended to model any arbitrary type of processor malfunction. It makes no assumptions on the behavior of faulty components. So it is the strongest failure model. If a DNS can tolerate Byzantine failures, it will be a hacker-tolerant system, i.e., it will continue to provide correct service even when some of its components are controlled by an attacker.

Performance

The performance of the new system should not degrade too much because of introducing the above new good properties. One design philosophy is to decrease the workload in the server side as much as possible. A resolver may send a request only once in a while, but a server may have to handle thousands of requests or more in one minute. Therefore, in some cases, we may have to trade the performance of clients for the performance of servers.

4.4 A Practical Byzantine-Fault-Tolerant Algorithm

4.4.1 Synchronous or Asynchronous

There is a lot of work on agreement and replication techniques that tolerate Byzantine faults [GM98, CR92, Rei96, KMMS98]. However, most of them either concern techniques that were designed to demonstrate theoretical feasibility, and which are too inefficient to be used in practice, or assume a synchronous system model. Assuming a synchronous system model means that the algorithms are relying on known bounds on message delays and process speeds to determine which replicas are faulty. However, a slow replica may be misclassified as faulty in these algorithms. Since correctness requires enough replicas to be non-faulty, a misclassification can compromise correctness by removing a non-faulty replica from the group. This opens an avenue of attack: an attacker gains control over a single replica but does not change its behavior in any detectable way; then he/she slows correct replicas or the communication between them until enough of them are excluded from the group.

A Byzantine-faulty-tolerant algorithm for the asynchronous system model does not make any assumption on the message delays and process speeds, so it is more practical and secure in the real world. This is what a Byzantine-fault-tolerant DNS system needs, for the Internet is highly asynchronous. But this kind of algorithms is very difficult to design. Actually the only practical one available now is the algorithm [CL99a, CL99b] invented by M. Castro and B. Liskov at MIT.

4.4.2 The System Model of CLBFT Algorithm

M. Castro and B. Liskov's new practical Byzantine-fault-tolerant algorithm, or CLBFT, assumes an asynchronous distributed system where nodes are connected by a network. The network may fail to deliver messages, delay them, duplicate them, or deliver them out of order.

CLBFT uses a Byzantine failure model, i.e., faulty nodes may behave arbitrarily, subject only to two restrictions:

1. Node failures are independent, i.e., a node failure does not necessarily cause other node failures.

2. Faulty nodes are still computationally bound so that they do not have the computation power to subvert the cryptographic techniques mentioned below.

CLBFT uses cryptographic techniques to prevent spoofing and replays and to detect corrupted messages. Messages in CLBFT contain public-key signatures [RSA78, RSA79], message authentication codes [Riv92], and message digests [Tsu98] produced by collision-resistant hash functions. All replicas know the public keys of other replicas to verify signatures.

CLBFT allows for a very strong adversary that can coordinate faulty nodes, delay communication, or delay correct nodes in order to cause the most damage to the replicated service. The only two assumptions are that the adversary cannot delay correct nodes indefinitely and it does not have enough computation power to break the cryptographic techniques used in CLBFT.

4.4.3 The Properties of CLBFT Algorithm

CLBFT can be used to implement any deterministic replicated *service* with a *state* and some *operations*. Clients issue requests to the replicated service to invoke operations and block waiting for a reply. The replicas are non-faulty if they follow the algorithm and if no attacker can forge their signature.

CLBFT provides both *safety* and *liveness* assuming that no more than $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty. *Safety* means that the replicated service satisfies *linearizability* [HW87] (modified to account for Byzantine-faulty clients): the service behaves like a centralized implementation that executes operations atomically one at a time.

Safety is provided regardless of how many faulty clients are using the service(even if they collude with faulty replicas), but it is insufficient to guard against faulty clients, e.g., in DNS a faulty client can send unauthorized update requests to the system. However, CLBFT limits the amount of damage a faulty client can do by providing access control: CLBFT can authenticate clients and deny access if the client issuing a request does not have the right to invoke the operation.

A *liveness* property is often informally understood as saying that some particular intended thing eventually happens. CLBFT does not rely on synchrony to provide safety. Therefore, it must rely on synchrony to provide liveness; otherwise it could be used to implement consensus in an asynchronous system, which is not possible [FLP85]. CLBFT guarantees liveness, i.e., clients eventually receive replies to their requests, provided at most $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty and $delay(t)$ does not grow faster than t indefinitely. Here, $delay(t)$ is the time between the moment t when a message is sent for the first time and the moment when it is received by its destination (assuming the sender keeps re-transmitting the message until it is received). This is a rather weak synchrony assumption that it is likely to be true in any real system provided network faults are eventually repaired.

Another important advantage of CLBFT is that it replaces some expensive public cryptography operations in authentication with message authentic codes (MACs). This makes this algorithm more than an order of magnitude faster than others, thus more practical.

CLBFT does not address the problem of fault-tolerant privacy: a faulty node may leak information to an attacker. Actually we do not need to consider privacy in DNS. It is part of the design philosophy of the DNS that the data in it are public and that the DNS gives the same answers to all inquirers.

In conclusion, given the good properties CLBFT has, it is clear that CLBFT is suitable for being used in DNS, which is a deterministic replicated system.

4.4.4 How the CLBFT Algorithm Works

CLBFT is a sophisticated algorithm; we will only sketch it here. Details of CLBFT can be found in [CL99a, CL99b].

CLBFT is a form of state machine replication: the service is modeled as a state machine that is replicated across different nodes in a distributed system. Each state machine replica maintains the service state and implements the service operations. There are at least $3f + 1$ replicas in the system, where f is the maximum number of replicas that may be faulty. We denote the set of replicas by $\mathcal{R}$.

The replicas move through a succession of configurations called *views*. In a view one replica is the *primary* and the others are *backups*. Views are numbered consecutively. The primary of a view is replica p such that $p = v \; mod \; \mathcal{R}$, where v is the view number. View changes are carried out when it appears that the primary has failed.

The algorithm works roughly as follows:

1. A client sends a request to invoke a service operation to the primary
2. The primary multicasts the request to the backups
3. Replicas execute the request and send a reply to the client
4. The client waits for $f + 1$ replies from different replicas with the same result; this is the result of the operation.

The CLBFT algorithm ensures the safety property by guaranteeing that all non-faulty replicas agree on a total order for the execution of requests despite failures.

When the primary replica receives a client request, it starts a three-phase protocol to atomically multicast the request to the replicas. The three phases are *pre-prepare*, *prepare*, and *commit*. The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary, which proposes the ordering of requests, is faulty. The prepare and commit phases are used to ensure that requests that commit are totally ordered across views. All the messages between replicas in this three-phase protocol are authenticated by message authentication codes.

The state of each replica includes the state of the service, a *message log* containing messages the replica has accepted, and an integer denoting the replica's current view.

The message log is kept for recovery purpose. Messages cannot be purged from a replica's log until it knows that the requests they concern have been executed by at least $f+1$ non-faulty replicas and it can prove this to others in view changes. Therefore, replicas need some proofs that the state is correct. These proofs are generated periodically, when a request with a sequence number divisible by some constant is executed. CLBFT refers to the states produced by the execution of these requests as *checkpoints* and a checkpoint with a proof is a *stable checkpoint.* A replica maintains several logical copies of the service state: the last stable checkpoint, zero or more checkpoints that are not stable, and a current state. A replica discards all old messages from the log and all earlier checkpoints after a *stable checkpoint* is generated. This is called *garbage collection.*

CLBFT has a view-change protocol to provide liveness by allowing the system to make progress when the primary fails. View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute. To make sure the view change mechanism cannot be used to mount denial-of-service attacks, at least $f+1$ replicas are required to trigger a view change.

4.5 Byzantine-Fault-Tolerant DNS

Using CLBFT, we designed and implemented a Byzantine-Fault-Tolerant DNS, or BFT-DNS.

4.5.1 System Model

In the BFT-DNS, the DNS name space structure and administration structure remains the same as in current DNS. We will discuss the way in which a resolver traverses the domain name space tree to get the final answer in Chapter 5 and Chapter 7. In the following sections we only discuss the basic operations, that is, a resolver sends a request to name servers of a zone and gets the answer without contacting name servers of other zones.

A client and the name servers of a zone compose an asynchronous distributed client-server system where nodes are connected by the Internet. Requests are sent, processed, and replied to in this system using a protocol based on CLBFT. This guarantees all the good properties we need in BFT-DNS (See Chapter4.3). Figure 4-1 shows a brief comparison between the current DNS with security extension and

BFT-DNS.

	current DNS	**BFT-DNS**
number of faults tolerated	none	f
number of servers in a zone	≥ 2	$\geq 3f+1$
number of zone authentication keys	one	$\geq 3f+1$
servers involved in a request	one	all
dynamic update	insecure	secure

Figure 4-1: Comparison of current DNS(with security extension) and BFT-DNS

Name Servers

There are at least $3f + 1$ name servers (*replicas*) in each BFT-DNS zone to provide correct service in the presence of up to f Byzantine server faults. Each server maintains a copy of the RR database of the zone. Their initial states should be identical. In contrast to the current DNS, in which one single key is shared by a whole zone to authenticate data RRs, each replica in BFT-DNS has its own public-private key pair. The public key of a replica is stored and authenticated in its parent zone or statically configured in a client. The private key is stored in the replica. So the replica can use the key to sign the data that will be sent to the client in real time. CLBFT also uses these keys to set up session keys between each pair replicas to protect messages between them from spoofing. Storing the key online will not cause the serious security problem that we face in the DNS security extension. If a replica is compromised, the intruder may capture the private key belonging to this replica and make it a faulty node. But he/she still will not be able to fake any convincing message under other replicas' names, i.e., the failure is localized to the compromised replica. Given that the number of faulty replicas is not larger than f, this BFT-DNS can still provide correct service. A private key can be stored in a smartcard to keep the attacker from stealing the key even if he/she breaks into the replica. This makes a faulty node recoverable, that is, after the intruder loses control over it. it will be able to recover by running a recovery protocol [CL99c].

Clients

Unlike in the current DNS, a client is not limited to be a resolver, which can only send read requests. It can also send update requests to name servers.

A client is statically configured with a list of $3f+1$ public keys that belong to the $3f+1$ name servers of a trusted zone. Then it can learn the public keys of the name servers of a zone by constructing an authentication chain, similar to the DNS security extension (see Chapter 3.3.3), but with a slightly difference. Each link in the chain must consist of the $3f+1$ public keys of each name server in a zone. Except for the beginning of the chain, which is statically configured, each link must be authenticated by at least $f+1$ (or $2f+1$ as we will discuss later) keys from the previous link. This authentication chain guarantees the authenticity of the keys at the end of chain, by which the final answer can be verified.

4.5.2 Read Schemes

Most of the requests sent to names servers are read-only requests, even in a highly dynamic zone. Since a read operation does not change the state of a name server, the only security issue is the authenticity of the answer. Based on this, we can optimize the 3-phase CBBFT algorithm to improve the performance of read-only operations. Below is the first read scheme.

Scheme 1

A resolver r first gets a list of IP addresses and public keys of the $3f+1$ name servers of the target zone. The authentication of the list is guaranteed by the authentication chain we mentioned above, given all involved zones are BFT zones. Then r sends $< query, t, (SK_i)_{K_i >}$ to every replica i in the list. Timestamp t is used to ensure that the reply of this query will not be replayed to do freshness attack. $(SK_i)_{K_i}$ is a session key between resolver r and replica i encrypted with the public key of r. Replicas execute the *query* immediately. They send the reply only after all requests reflected in the tentative state have committed; this is necessary to prevent the client from observing uncommitted state. A reply from i includes $< result, t >$ and its digest D_i, using SK_i as the digest key. The resolver checks D_i to see whether the reply is truly from i. Since only r and i know SK_i, no one can modify the reply without r's notice. r will not be fooled by a correct answer to a modified question, because the *result* already includes the original *query*. The *result* need not include any SIG RRs or NXT RRs, for the D_i already proves its authenticity. r waits for $2f+1$ replies from different replicas with the same result. r may not be able to collect $2f+1$ such replies if there are concurrent writes to data that affect the result;

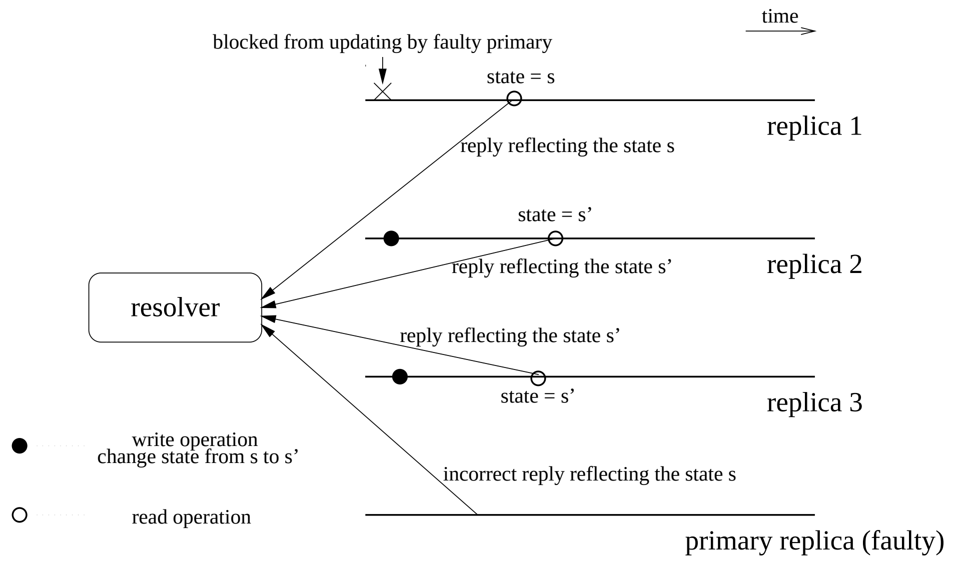

time
blocked from updating by faulty primary
state = s
replica 1
reply reflecting the state s
state = s'
replica 2
reply reflecting the state s'
resolver
reply reflecting the state s'
state = s'
replica 3
write operation
change state from s to s'
incorrect reply reflecting the state s
read operation
primary replica (faulty)

Figure 4-2: $f + 1$ identical replies are not enough in the optimized read scheme

in this case, it retransmits the request using a scheme similar to the update scheme (described below) after timer expires.

$f + 1$ replies from different replicas with the same result are not enough for r to conclude the correct answer. In case the primary replica is faulty, up to f non-faulty replicas may be blocked from receiving update requests. They may carry stale data for a long time without notice. Figure 4-2 shows an example. In this example, the faulty replica and replica 1 both send replies reflecting the stale state s. If the resolver waits only for $f + 1$ identical replies, it will accept a stale answer.

Setting up a session key between a resolver and a replica for each request is quite expensive. This can be optimized by caching session keys in both sides for future use. Section 5.4.6 will discuss it in detail.

Scheme 2

If the session key caching mechanism does not work well, scheme 1 is relatively slow. Although it provides the most secure service, it may be a little too expensive when median-level security is already enough for the client. For this reason, we designed another read scheme, which can improve the performance of servers, while still providing service more secure than the DNS security extension.

In the second scheme, a name server must use its private key to pre-compute SIG RRs for each RR set (RRs with the same domain name, class and type) in its database, just like in the DNS security extension, but in a slightly different way. A SIG RR is calculated as $< RRs, T_{generate}, T_{expire}, SN_{database} >_{sig}$ That is, a signature record covers not only the corresponding RR set, the generation time and the expiration time, but also the database serial number that is included in the SOA RR. The SOA SIG RR is re-generated in a fixed short interval. Once generated, a SOA SIG RR expires very quickly, just shortly after the next generation (we need to add a time margin for transmitting the SOA SIG RR to a client). If any RR (excluding the SIG RRs) in the replica database changes, for example, after an update, the database serial number in the SOA RR must be incremented immediately and all SIG RRs in this zone must be re-calculated according to the new database serial number.

To do a query using scheme 2, a resolver r first gets a list of IP addresses and public keys of the $3f + 1$ name servers of the target zone. Then r sends $< query >$ to every replica i in the list. Replicas execute the *query* immediately, then send back the result. The result includes: the requested RRs and corresponding SIG RRs, SOA RR and SOA SIG RR. r uses the public key of a replica to verify the authenticity of received RRs. If the SOA SIG RR has not expired yet, it means the database serial number SN in the SOA RR is *almost up-to-date.* By checking whether the SN in the SOA RR agrees with the SN covered by other SIG RRs, r can know whether the received RRs, which answer r's query, are from an *almost up-to-date* version of database. r waits for $2f + 1$ replies from different replicas with the same result to conclude the answer.

Scheme 2 need no per-query-based encryption and decryption operations on the servers' side. So it can dramatically decrease the workload of name servers. It is especially important for some hot zones, which may receive thousands of queries or more per seconds. Scheme 2 trades off the security for performance. It is less secure than scheme 1, but seems secure enough in most cases in the real world. More details are discussed in Section 4.6. One problem with scheme 2 is that every replica has to re-generate all SIG RRs even if only one RR (excluding SIG RRs) changes. This process is quite expensive if the zone is large. So this scheme only suits zones that are small and are not updated very often.

RR Caching

RR Caching is a very important property of today's DNS. If queries are resolved recursively (see Section 2.5), a name server may cache the RRs that it received from other name servers so that later queries for the same RRs can use the cached result and will thus avoid additional queries to other servers. RR caching is only useful for read operations.

In both BFT-DNS read schemes, RR caching is not allowed if users always require the freshest data, since the cached data may be stale. But abandoning the caching mechanism may cause the name servers of the top zones, especially the name servers of the *root* zone, to be overloaded. To solve this problem, we can allow some caching. A name server can set some flags in its reply packet to indicate whether the RRs in its reply packet is cacheable or not. Also when a client sends a query to the name servers, it can use a flag to indicate whether cached data is acceptable.

In scheme 1, when cached RRs are sent to clients, they are authenticated by the replicas in just as the same way that is used to authenticate normal RRs in databases, that is, using the session keys to generate message authentication codes.

In scheme 2, if a name server wants to cache some RRs (target RRs), it must also cache the corresponding SIG RRs and SOA SIG RR. These RRs will be used to prove the authentication of the cached target RRs later. Although in most cases the cached SOA SIG RR will expire when a client receives it, the client still can use it to determine the freshness of the cached target RRs.

We will discuss the RR caching mechanism in BFT-DNS further in Section 7.4.

4.5.3 Update Scheme

Generally, update requests are far less common than read requests, but they are more critical.

In BFT-DNS, an update request is handled as a typical transaction in the CLBFT algorithm.

We denote a message m signed by node i as $< m >_{\sigma_i}$. A client c requests the execution of update operation o by sending a $< o, t, c >_{\sigma_c}$ message to the primary replica of the target zone. Timestamp t is used to ensure *exactly-once* semantics for the execution of client requests. Timestamps for c's requests are totally ordered such that later requests have higher timestamps than earlier ones; for example, the timestamp could be the value of the client's local clock when the request is issued.

Notice that the idea of the *primary replica* in BFT-DNS, which is determined by the current *view number*, is completely different from the idea of the *primary name server*, which is listed in the zone SOA RR in the current DNS.

A client sends a request to what it believes is the current primary (this information may be piggybacked in some message received earlier) using a point-to-point message. The primary atomically multicasts the request to all the backups and triggers the 3-phase protocol, which was described in Section 4.4.4.

A replica sends the reply to the request directly to the client. The reply has the form $< reply, v, t, c, i, r >_{\sigma_i}$ where v is the current view number, t is the timestamp of the corresponding request, i is the replica number, and r is the result of executing the requested update operation.

The client waits for $f + 1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r. This ensures that the result is valid, since at most f replicas can be faulty. If the result is a positive commit, the client knows that its update request has been persistently applied to the target zone, and visible to any further queries, despite up to f Byzantine node being faulty. This is guaranteed by the good properties of CLBFT.

If the client does not receive replies soon enough, it broadcasts the request to all replicas. If the request has already been processed, the replicas simply re-send the reply; replicas remember the last reply message they sent to each client. Otherwise, if the replica is not the primary, it relays the request to the primary. If the primary does not multicast the request to the group, it will eventually be suspected to be faulty by enough replicas to cause a view change.

Unlike in the read schemes, replicas must check the authenticity and authority of the incoming update requests in the update scheme. BFT-DNS uses the following authentication and authorizations policy.

1. An update request signed by a private key belonging to a domain name has the authority to update any RRs owned by this domain name.

2. An update request signed by a special *zone master key* has the authority to update any RRs in the zone. This *zone master key* should be different from any replica's key that is stored online. The public part of the zone master key is known to every replica while the private part should be stored off-line and only accessible to the zone administrator. Considering the fact that the zone master key is only used to sign update requests, and the RRs data are constantly protected by the keys of $3f + 1$ replicas, storing the zone master key offline will not cause the security problems presenting in the current DNS security extension as we have mentioned, but some inconvenience. A practical way to do update is by signing update requests in the isolated host that holds

the zone master key, then using floppy disk to copy the signed requests to a network-linked host from where the requests are sent to the target zone.

4.6 Attacks to the New DNS

The BFT-DNS is not perfect, it still has some security holes.

Flooding attack

In the current IP protocol (IP version 4), there is no IP level packet authentication. In addition, BFT-DNS does not have any read access control. Therefore BFT-DNS is still vulnerable to flooding attacks. Malicious users can send thousands of useless requests to the target servers to slow them down and to eventually cause other resolvers timeout.

This problem can be partially solved as following. After receiving a request packet, the server sends a packet containing a random number back to the IP address the requester claimed. The server will not start processing the request before receives a packet containing the random number it just sent. This trick makes it difficult for a remote attacker to hide the original IP, if he/she can control any router used by the server. As a result, the malicious user's IP address may be identified and be blocked from further service. But it will not eliminate attacks from the LAN or a hacker router used by the server.

The ultimate solution is to provide a very low level (e.g. packet level) authentication protocol, which is beyond the scope of this thesis.

Freshness attack

Read scheme 2 is still vulnerable to freshness attacks. Attacker can replay an old reply set before the SOA SIG RR expires. But the lifetime of a SOA SIG RR can be very short, for example, 3 seconds, so the attack window is very small. In the real world, 3-second-stale DNS information should not cause any problem in most cases.

Chapter 5

Implementation

5.1 Overview

This chapter gives an overview of the implementation of BFT-DNS. The full BFT-DNS implementation is based on *TIS/DNSSEC* [TIS99], a beta version of DNS implementation with security extensions developed by Trusted Information Systems, Inc. This chapter mainly focuses on the issues relating to the implementation of Byzantine-fault-tolerant features of the new system.

This chapter starts with a description of a BFT replication library, which plays the key role in our implementation, followed by an overview of the implementation of the *TIS/DNSSEC*, on which our work is based. Then we will describe the architecture of our BFT-DNS and discuss each module in BFT-DNS. Finally, we will describe the limitations of our BFT-DNS implementation.

5.2 The BFT Replication Library

M. Castro and B. Liskov developed and publicized a replication library, which can be used as a basis for any Byzantine-fault-tolerant replicated service. We use this library in the implementation of BFT-DNS.

The client interface to the replication library consists of a single procedure, *invoke*, with one argument, an input buffer containing a request to invoke a state machine operation. The *invoke* procedure uses the CLBFT protocol to execute the requested operation at the replicas and select the correct reply from among the replies of the individual replicas. It returns a pointer to a buffer containing the operation result.

On the server side, the replication code makes a number of upcalls to procedures that the server part of the application must implement. There are procedures to execute requests (*execute*), to maintain checkpoints of the service state (*make_checkpoint*, *delete_checkpoint*), to obtain the digest of a specified checkpoint (*get_digest*) and to obtain missing information (*get_checkpoint*, *set_checkpoint*). The *execute* procedure receives as input a buffer containing the requested operation, executes the operation, and places the result in an output buffer.

Point-to-point communication between nodes is implemented using UDP, and multicast to the group of replicas is implemented using UDP over IP multicast. There is a single IP multicast group for each service, which contains all the replicas. These communication protocols are unreliable; they may duplicate or lose messages or deliver them out of order. CLBFT can tolerates out-of-order delivery and rejects duplicates. View changes can be used to recover from lost messages, but this is expensive and therefore it is important to perform retransmissions. During normal operation recovery from lost messages is driven by the receiver: backups send negative acknowledgments to the primary when they are out of date and the primary retransmits the pre-prepare message after a long timeout.

The replication library does not implement view changes or retransmissions at present.

5.3 TIS/DNSSEC

Instead of writing a BFT-DNS package from scratch, we used the TIS/DNSSEC (version BETA-1.4) as the basis for the BFT-DNS implementation.

TIS/DNSSEC is developed by Trusted Information Systems, Inc., as an important step in their effort towards creating a new generation of secure DNS.

We chose TIS/DNSSEC because it is the only available DNS package that implements (but not fully) the DNS security extension. Although in our read scheme 1 we do not need SIG RRs to authenticate data, we still depend on SIG RRs in our read scheme 2, so using TIS/DNSSEC as our basis saved us a lot of time.

Another reason to choose TIS/DNSSEC is its compatibility. TIS/DNSSEC is based on the Berkeley Internet Name Domain (BIND, version 4.9.4-p1) [AL98], which implements an Internet name server for BSD-derived operating systems. BIND is estimated to be the DNS software in use in over 90% of the hosts on the Internet,

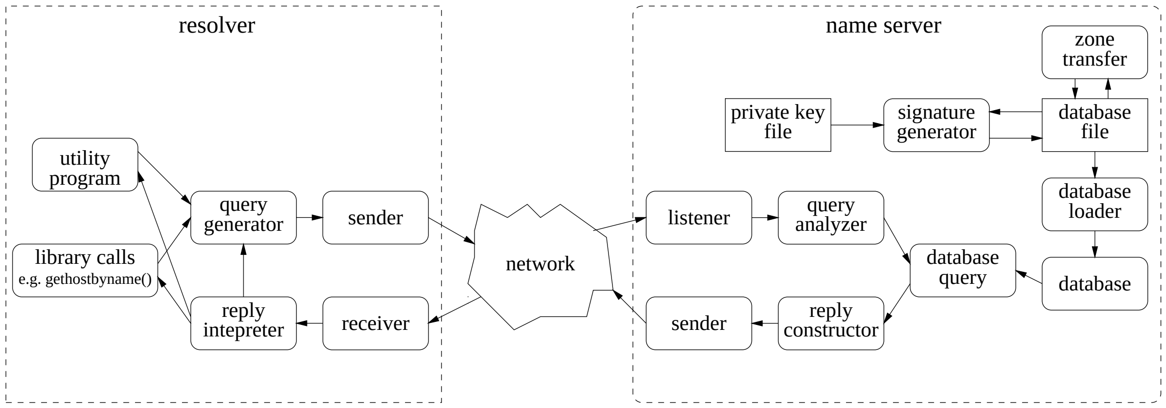


Figure 5-1: Architecture of TIS/DNSSEC

and is highly portable. As a result, TIS/DNSSEC can run in most UNIX operating systems. It is also easy to migrate TIS/DNSSE to Windows NT.

The TIS/DNSSEC consists of a name server (or "daemon") called *named*, a resolver library, and several utility programs. Its architecture is shown in Figure 5-1

Compared to the BIND, TIS/DNSSEC made some modifications to add DNS Security features.

1. A signature generator is added. This generator reads in RRs from zone database file, generates the corresponding SIG RRs and NXT RRs using the private key stored online, then adds these RRs to the database file. RSA PPK algorithm [RSA78, RSA79] is used for the signing process.

2. The database query module does not only return the asked RRs for a query, but also adds the corresponding SIG RRs to its answer set. If there is no data for a query, a proper NXT RR and NXT SIG RR will be returned, instead of a NULL answer.

3. The zone transfer module, database load module and reply construct module can handle the new added RR types, such as SIG RR and NXT RR, correctly.

4. The reply interpreter module at the client side can interpret the new added RR types correctly. Some enhanced client tools are provided. For example, a tool named *dig* can dump all types of RRs in a reply to screen in human-understandable ASCII text.

The DNS security extension is only partially implemented. TIS/DNSSEC does not support the dynamic update operation. The only way to update the zone data is to edit the database file, use the signature generator to sign the data, then reload the database to the memory.

TIS/DNSSEC also does not provide any signature verifier for the client side, at least in the version BETA-1.4, which seems very strange. Without a signature verifier, it is even worse than a non-secure DNS. Firstly it is slower. Furthermore, people who use this “secure” DNS may think the results they got are authenticated, but they are not.

Resolvers communicate with name servers by UDP in TIS/DNSSEC. If a query or reply packet is lost, the sender module at the client side will re-transmit the query packet after timeout.

5.4 BFT-DNS

Simply speaking, we build the BFT-DNS by combining TIS/DNSSEC and the BFT Replication Library together.

Figure 5-2 shows the architecture of BFT-DNS.

In the following subsections we will describe the modules that compose the system one by one.

5.4.1 CLBFT Replication Library

The sender module and receiver module in the client side are replaced by CLBFT replication library calls. The query generator module calls the *invoke* procedure of the replication library, and the result is sent back to reply interpreter module.

Also the listener module and sender module at the server side are replaced by CLBFT replication library calls. An authenticated incoming request will trigger the *execute* procedure, which will call the database update module or the database query module, depending on the request type.

We made several modifications to the CLBFT replication library to fit it to the BFT-DNS.

1. The BFT-DNS servers never verify the authenticity of a read request. Making the DNS data accessible to everyone is one of the DNS designing philosophies. So the authentication check is disabled in CLBFT replication library for read requests from clients. It will not weaken the security of the system, if we do not take flooding attacks into consideration.

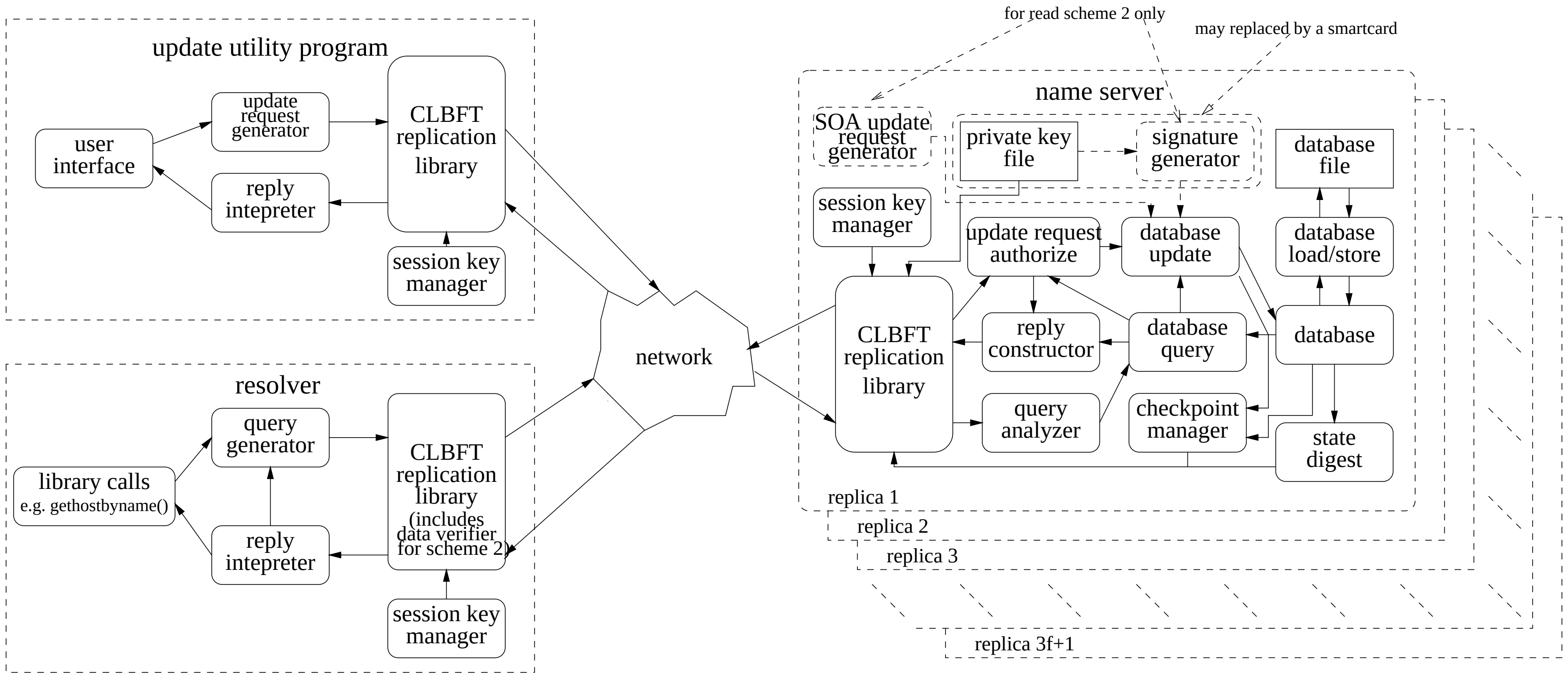


Figure 5-2: Architecture of BFT-DNS

2. In read scheme 2, for performance reasons, replicas do not add per-query-based authentication information to the replies sent to clients. We modified the CLBFT replication library so that in read scheme 2, the authentication and freshness of a replica reply is verified by the pre-generated signatures for each RR, instead of one signature calculated in real time for the whole reply message. Also in scheme 2, the term "same result" is defined as the result containing the same requested RRs and SOA RR, excluding any SIG RRs, which are surely different from each other if they are from different replicas.

3. The procedure of establishing a session key between a client and a replica has not been implemented in the CLBFT replication library. We add this part to the library.

4. In original CLBFT, a client uses a group IP address to multicast a read-only request to all replicas. It is faster than unicasting but may not be widely supported in the Internet. So we change it to send a read-only request to the replicas one by one. The communications between replicas still use multicasting.

5.4.2 Modules From TIS/DNSSEC

Query Generator Module and Reply Interpreter Module

They are modified so that the CLBFT replication library can get the correct $3f + 1$ public keys to verify data for each query. That is, in each round of an iterative query resolution, the reply interpreter module must extract the public keys from the reply and pass them to the CLBFT replication library for the next round.

Reply Constructor Module and the Query Analyzer Module

We do not change these two modules a lot, except that the reply constructor module now can generate a positive or negative reply to inform the client that the update request was committed or aborted.

Database Load/Store Module

Although BFT-DNS can tolerate up to f faults, saving the database to the persistent storage after modification is still necessary. It is possible that all servers lose their states in the memory at the same time, for example, a total power loss in a building.

To solve this problem, we can back each replica with an independent UPS, and save the state in primary memory to persistent storage if a power loss happens. We have not modified this module so the storing has not been implemented yet.

Database Query Module

We made a small change to this module. In read scheme 2, the SOA RR and SOA SIG RR are passed to the reply constructor for each query, in addition to the RRs and corresponding SIG RRs the client requested.

Database Module

The structure of the database remains the same: all RRs that have the same domain name are stored in a sorted link list. Pointers to these lists are stored in a hashtable using hashed domain names as keys. The only difference is that we add a *copy on write bit*, a *ghost bit* and a hash value for each RR entry, and a *copy on write bit* for each domain name entry. We will describe these later.

Signature Generator Module

This module is used only if read scheme 2 is used.

We made a little modification to this module so that the current database serial number, which is stored in the SOA RR, is digested and covered in every SIG RR generated by this module.

This module can be integrated into a smartcard, which also stores the private key of a replica. This can prevent an intruder from stealing the private key even if he/she can break into a replica.

5.4.3 Some New Modules

Update Request Generator Module

In this module, an update request message is generated according to the user's request. There are three types of update requests: add, delete, and modify. Since there can be multiple RRs with the same name, class, and type in one zone, the delete or modify request should include the entire RRs, not just names, classes and types, that need to be deleted or modified. This request will be passed to the CLBFT replication library, together with the proper private key that has the authority to do this update. The

CLBFT replication library will generate a signature that shows the authenticity and authority of this request.

In a safer mode, we can pass an update request and a corresponding pre-generated signature, which is transmitted from an isolated host that holds the proper private key, to a ‘shortcut’ point in the CLBFT replication library. Since the private key is not present in any machine that is linked to the network, a remote hacker has no way to steal it.

Update Request Authorize Module

This module holds the update authority policy, which we described in Section 4.5.3. It cooperates with the CLBFT replication library to check the authenticity and authority of the incoming update request. It interprets the update request, fetches the proper public key from database, then passes the key back to the CLBFT replication library.

SOA SIG Update Request Generator Module

This module is only used if read scheme 2 is used. A simple timer will send an SOA SIG update request to the database update module in a fixed interval. The value of this interval is affected by 2 things: the level of security desired and the computation power of a replica. Timers in different replicas need not to be synchronized. But the clocks in replicas and resolvers must be synchronized; otherwise a resolver may always decide that an answer has expired when actually it has not.

Database Update Module

This module adds, deletes, or modifies the specified RR entries in the database according to the update request.

If read scheme 2 is used, the following actions must be taken after the requested update operation is finished;

1. Update related NXT RRs.
2. Increase the serial number in the SOA RR.
3. Delete all old SIG RRs and re-generate them based on the new database serial number.

To generate the proper checkpoints, the database update module must cooperate with the checkpoint manager module. This module will be described in the next section.

5.4.4 Checkpoint Manager Module

The checkpoint manager module generates and maintains checkpoints of the state of the RRs database. Each replica maintains several logical copies of the state: the last stable checkpoint, zero or more checkpoints that are not stable, and a current state. These states are kept for recover purpose.

We use copy-on-write to reduce the space and time overhead associated with maintaining checkpoints. This technique is also used in a Byzantine-fault-tolerant file system, or BFS, developed by M. Castro and B. Liskov, but our approach is slightly different. In BFS, the number of the file blocks is fixed, but in DNS, the number of RRs can vary. This difference also affects the state digest module, which we will describe in the next section.

Here are the details of the copy-on-write technique. The database always maintains the current state. Every name entry and RR entry in the database has a copy-on-write bit. When the replication code invokes the *make_checkpoint* upcall, we set all the copy-on-write bits to 1 and create a checkpoint record containing the current sequence number, which it receives as an argument to the upcall, and a list of RRs. This list contains the copies of the RRs that have been modified since the checkpoint was taken, and is initially empty. The record also contains the digest of the current state; the computation of the digest will be discussed in the next section.

When the database update module wants to delete an RR entry from the database, it first stores this RR in the checkpoint record for the last checkpoint, and then it sets the *ghost bit* for this RR entry in the database to indicate this RR has been deleted. The associated hash value of this RR is untouched. The copy-on-write bit of the corresponding name entry is set to 0.

When the database update module wants to add an RR entry to the database, the copy-on-write bit of this entry and the copy-on-write bit of the corresponding name entry (in the domain name hashtable) are set to 0. The *ghost bit* is set to 0 and the hash value is set to null. This RR is also copied to the checkpoint record for the last checkpoint, with a special "existence" flag set to indicate that this RR did not exist before this checkpoint.

When the database update module wants to modify an RR entry in the database, it first checks the copy-on-write bit of this RR entry. If it is 1, we copy this RR to the checkpoint record for the last checkpoint, then set its copy-on-write bit and the copy-on-write bit of the corresponding name entry to 0. Finally the RR is overwritten with the new value.

The checkpoint manager module retains a checkpoint record until the module is told to discard it via a *delete_checkpoint* upcall, which is made by the replication code when a later checkpoint becomes stable.

If the replication code requires a checkpoint to send to another replica, it calls the *get_checkpoint* upcall. To obtain the value for an RR in the last stable checkpoint, we first search for the RR in the checkpoint record of the stable checkpoint, and then search the checkpoint records of any later checkpoints. If the RR is found in some checkpoints with the "existence" flag set, which means the RR was added after the last stable checkpoint, a negative reply will be returned. If the RR is not found in any checkpoint record, it means that this RR has not been modified since last stable checkpoint. In this case, we search the database to return the required RR value in the current state.

We do not consider any SIG RR or NXT RR as a part of the state of a replica, because they are totally derived from the other RRs and are useless to other replicas. So checkpoints do not reflect any changes in SIG RRs or NXT RRs. As a result, although all SIG RRs are modified in an update operation if we use read scheme 2, only the 'basic' updated RRs, which exclude SIG RRs and NXT RRs, will be added to the checkpoint record. Also the frequent SOA SIG update requests will not cause the checkpoint record to grow.

5.4.5 State Digest Module

This module computes a digest of a checkpoint state as part of a *make_checkpoint* upcall. Although checkpoints are taken only occasionally, it is important to compute the state digest incrementally because the state may be large. We uses an incremental collision-resistant one-way hash functions called AdHash [BM97], which is also used in M. Castro and B. Liskov's BNS in a slightly different way. This function divides the state into some unique blocks and uses some other hash function to compute the digest of each block. The digest of the state is the sum of the digests of the blocks modulo some large integer. In our current implementation, we use each RR as a block unit and compute its digest using MD5 [Riv92]. Again, the SIG RRs and NXT RRs are excluded from the state computation, because the digest of all 'basic' RRs is enough to prove the integrity of the state.

To compute the digest of the state incrementally, we store a hash value for each RR in the database. This hash value is obtained by applying MD5 to the RR at the time of last checkpoint. When *make_checkpoint* is called, we obtain the digest d for the previous checkpoint state from the associated checkpoint record. We then traverse the table that contains pointers to linked lists of RRs for each domain name. If the *copy on write bit* of a table entry is set to 0, we traverse the corresponding linked list and check the *copy on write bit* of every RR in this list to find the updated RRs. For each updated RR, we do the following operations.

1. Subtract the old hash value, which is stored in the updated RR entry, from d. If the old hash value is null, skip this step.

2. Compute new hash value for the updated RR, add it to d. If the *ghost bit* of this entry is set, skip this step.

3. If the *ghost bit* of this entry is set, delete this entry from the link list, otherwise update the RR entry to contain the new hash value.

This process is more efficient than digesting the whole state, provided the number of modified RRs is relatively small compared to the number of RRs in the database.

5.4.6 Session Key Management Module

This module is only used in read scheme 1 and in the update scheme. It caches the session keys used in previous transactions to decrease the workload in both the clients and the servers and therefore improves performance.

To set up a session key between a client and a replica is very expensive. Actually it is the dominant part of a transaction. So we store the session key at both client side and server side for further use after a successful transaction. Session keys are stored in hashtables. We use hashed IP addresses as entry keys. This is based on two reasonable assumptions:

1. Applications in a client host will share one resolver and totally trust it.

2. Every server host runs only one name server process.

When a client wants to send a request to a server, it first checks its session key table to see whether it has talked to this server before and if there is a valid session key between them. If so, it sets a flag in the request message to tell the server to use the session key they used before. If the server can find the key in its table, it uses this session key to sign the reply message. If the server cannot find the session key in its table, it will generate a new session key, send back the encrypted session key to the client together with the reply message that was signed by the session key. The client decrypts the session key, uses it to verify the reply message, and updates the session key table.

Notice that a session key that is set in a read transaction cannot be used in an update transaction. Unlike a session key for read transactions, to establish a session key for update transactions, a client must provide a valid signature of an authorized private key. We keep session keys for read transactions and session keys for update transactions in two different hashtables.

Since a session key never appears in the network in plain text, a hacker cannot know the key unless he/she breaks into the server host or client host. If a server host is broken into, we treat it as a faulty node and can tolerate it if the number of faulty nodes is less than f. If a client host is broken into, the session key becomes useless because the intruder is already able to do everything in the hacker host.

An attacker may use a faked IP address to set up a session key for read transactions with a server. This attack will degrade the system but will not cause a client to be fooled by a wrong answer. A session key in CLBFT is only used for authentication purpose. Also the client and the server will eventually find the key discrepancy between them and re-establish a session key.

Each entry in the session key table contains a 128-bit session key, a 32-bit IP address, and some statistical information. An entry replacement algorithm uses the statistic information to determine which entry to evict when the hashtable is too crowded. The replacement algorithm can vary in different machines but it should be based on the following facts.

- A client that visited a name server recently will be more likely to visit it again in the near future. (Time locality)
- The more times a client visited a name server, the bigger the probability that it will contact that server again.

- A client is more likely to send requests to a local or top domain name server. (Space locality)

5.5 Implementation Limitations

Our biggest limitation is that we have not implemented the view change and failure recovery, because the BFT Replication Library does not support this yet. This limitation does not affect the correctness of our system, that is, the client will never accept a wrong answer, but it does affect the liveness of our system.

Another big limitation is that we have not implemented packet re-transmission yet. In our implementation all communication is carried by UDP packets, which are vulnerable in the always-congested Internet. In our design, a client needs to collect $2f + 1$ identical replies for a read request and $f + 1$ replies for an update request from the $3f + 1$ replicas. If some request or reply packets are lost, a client may not be able to collect sufficient identical replies to finish the query. Replacing UDP with TCP may not be a good solution, since TCP is more expensive. Our proposal is to let a client time out and re-send its request if it cannot collect enough authenticated replies.

Due to this limitation, we can only do the performance tests on our local network, in which UDP packets are almost never lost.

Chapter 6

Performance Evaluation

6.1 Overview

This chapter describes the performance evaluation of our Byzantine-fault-tolerant DNS, or BFT-DNS. We will compare BFT-DNS with TIS/DNSSEC, which is an implementation of DNS with the security extension. We will also compare BFT-DNS with an implementation of DNS without the security feature. Figure 6-1 shows brief descriptions of the systems tested in our experiments. We denote the BFT-DNS scheme 1 as BFT-DNS-1, and the BFT-DNS scheme 2 as BFT-DNS-2.

Test System	Authentication Means
TIS/DNS	no authentication means
TIS/DNSSEC	using pre-generated SIG RRs signed by the zone private key
BFT-DNS-1	using messsage authentication codes (MACs) generated by the session key that was established by using the server public-private key pair
BFT-DNS-2	using SIG RRs signed by the server private key + generating SOA SIG RR with short TTL in a short interval to prove the freshness of the database

Figure 6-1: Brief Description of the Systems Tested in Our Experiments

We will show that for queries BFT-DNS-1 performs as well as or even better than TIS/DNSSEC if the session key is cached, but is very slow if the session key is not in the cache. BFT-DNS-2 is slightly slower than TIS/DNSSEC, but its performance is still acceptable. Update operations in BFT-DNS-1 are fast enough for most zones, while in BFT-DNS-2, update operations are only practical in small and relatively stable zones.

The rest of this chapter is organized as follows. Section 6.2 describes our experimental setup. Section 6.3 describes the experiments. Section 6.4 presents the results of our experiments that measure the performance of BFT-DNS relative to the performance of the TIS/DNSSEC and the DNS without security feature. Section 6.5 discusses some other performance issues. Section 6.6 compares BFT-DNS-1 and BFT-DNS-2,

6.2 Experimental Setup

The experiments measure normal-case behavior (i.e., there are no view changes) in BFT-DNS, because this is the behavior that determines the performance of the system.

All experiments were run with one client and four replicas, which can tolerate one Byzantine fault.

Replicas and client ran on identical DEC 3000/400 Alpha workstations, each with a 133 MHz Alpha 21064 processor, 128 MB of memory, and OSF/1 version 4.0B. The RR databases were all loaded into memory, so that there were no disk operations involved in the experiments. The machines were connected by a 10Mbit/s switched Ethernet and had DEC LANCE Ethernet interfaces. The switch was a DEC EtherWORKS Switch 8T/TX. The experiments were run on an isolated network.

The interval between checkpoints was 128 requests, which caused garbage collection to occur several times in the experiments for update requests. The maximum sequence number accepted by replicas in pre-prepare messages was 256 plus the sequence number of the last stable checkpoint.

The modulus length of the RSA keys that we used in the experiments was 512 bits. The public exponent of keys was 3. Compared to 65537, another possible value for key public exponent, using 3 as the public exponent makes verifying operations and encrypting operations much faster (10 times), while signing and decrypting speed remains the same.

6.3 Experiments

To compare with our BFT-DNS, we added a reply authentication verification module to TIS/DNSSEC. This module checks the SIG RRs in the replies to verify the authenticity of the returned RRs.

TIS/DNSSEC is based on BIND version 4.9.4-p1, which is the most popular DNS package. We disabled the security features in TIS/DNSSEC and recompiled it to get a version called "TIS/DNS". We used TIS/DNS to test the performance of a DNS without security features.

There is no standard benchmark for DNS, so we had to design the test ourselves. There are 3 types of requests tested in our experiments.

A RR Query Query for the IP address of host *host1.foo.bar.* Queries are sent to the name servers of zone *foo.bar.* One A RR and an optional SIG RR will be returned to the client. This type of query is the simplest. It is also one of the most frequently used query types in the real world. Actually, it is the original driving force for the creation of the DNS infrastructure.

NS RR Query Query for name servers of zone *foo.bar.* Queries are sent to the name servers of zone *bar.* In TIS/DNSSEC or TIS/SEC, there are 2 name servers for this zone, which is a typical setting. In BFT-DNS, there are 4 name servers for this zone. Generally, this type of query may return lots of RRs, including NS RRs, A RRs for name servers, related KEY RRs, and optional SIG RRs. It is one of the most complicated query types. This type of query is also one of the most frequently used query types in the real world. Resolvers must know the name servers of the target zone before they do any queries to this zone. In most cases, this information is collected from other name servers by one or several NS RR queries.

A RR Update Request The only type of update request tested in our experiments is to change the IP address of a host in zone *foo.bar.* This is the simplest update request type.

Since the zone databases are all loaded into memory and we use hash tables to store the database records, the core database lookup procedure takes very little time regardless of the size of the databases. As a result, we did tests only in small zones containing about 120 RRs.

6.4 Results and Explanations

We will show the test results for different types of requests in this section. All test data were obtained by timing 10,000 operation invocations in three separate runs and we only report the median value of the three runs. The maximum deviation from the median was always below 1% of the reported value.

In this section we assume session keys have been established when we test BFT-DNS-1. In most cases this is a reasonable assumption if the session key cache mechanism works well (see Section 5.4.6, 7.2, and 7.3). We will discuss the cost to establish a session in Section 6.5.1.

In the end of this section, we will use the test results to evaluate each system.

6.4.1 Query for the IP Address of a Host

Figure 6-2 shows the results of the tests of A RR queries. The *elapsed time* is counted from the time the client starts to generate a request to the time it accepts a correct reply and interprets the reply successfully. The *server time* is counted from the time the server receives the request packet to the time the reply packet is ready to be sent. That is, it does not include the time to receive requests or the time to send out replies. The *server CPU* is the host CPU usage of the name server process during the experiments.

Test System	**#RR/ #SIG**	**Reply size(byte)**	**Elapsed Time (ms)**	**Server Time (ms)**	**Server CPU**
TIS/DNS	1/0	47	1.03	0.122	38.1%
TIS/DNSSEC	2/1	150	2.96	0.184	18.0%
BFT-DNS-1	1/0	112	2.18	0.205	19.8%
BFT-DNS-2	4/2	312	13.1	0.365	5.08%

Figure 6-2: Query for the IP address of a Host

In Figure 6-2, the *elapsed time* in TIS/DNSSEC is longer than in TIS/DNS. The difference is introduced by the SIG RR verification. Our tests show that each SIG RR verification costs about 1.75ms, all at the client. A larger reply packet (150 bytes vs. 47 bytes) contributes to not only a longer *elapsed time* but also a longer *server time*.

The *elapsed time* and *server time* in BFT-DNS-1 are longer than in TIS/DNS. The overhead is introduced by the replication library, which has extra computation and uses larger request/reply packets. The cryptographic operations for each request cost about 0.3ms. This is significantly smaller than the corresponding cost in TIS/DNSSEC. The reply packet is also smaller (112 bytes vs. 150 bytes). As a result, the *elapsed time* in BFT-DNS-1 is smaller than in TIS/DNSSEC.

In BFT-DNS-2, since we need to verify 2 SIG RRs for each reply from a replica and we need to collect $2f + 1 = 3$ verified and identical replies to get a final answer, 6 SIG RRs must be verified for each request. It costs 10.5ms in all. These expensive operations plus the use of a large reply packet (312 bytes) causes the *elapsed time* to increase to 13.1ms compare to BFT-DNS-1. The bigger reply packet also causes the *server time* to grow.

6.4.2 Query for Name Servers of a Zone

Figure 6-3 shows the results of the tests of NS RR queries.

Test System	**#RR/ #SIG**	**Reply size(byte)**	**Elapsed Time (ms)**	**Server Time (ms)**	**Server CPU**
TIS/DNS	4/0	98	1.37	0.312	40.9%
TIS/DNSSEC	9/4	592	8.76	0.457	10.3%
BFT-DNS-1	12/0	552	3.45	0.673	27.3%
BFT-DNS-2	23/10	1567	59.9	1.05	2.42%

Figure 6-3: Query for Name Servers of a Zone

Because of bigger answer sets, the *elapsed time* and the *server time* with NS RR queries are bigger than with A RR queries in every system.

The *elapsed time* of TIS/DNSSEC increases from 2.96ms to 8.76ms, since the number of SIG RRs increases from 1 to 4. The four SIG RR include one for the NS RRs, one for the zone KEY RR, and one for each name server's A RR. In contrast, the *elapsed time* of BFT-DNS-1 does not increase substantially (from 2.18ms to 3.45ms), because a client only needs to verify one MAC for each reply, regardless how many RRs are included.

In BFT-DNS-2, a client needs to verify $(2f+1)\times 10 = 30$ SIG RRs for each request. Each reply from a replica includes one SIG RR for the SOA RR, one SIG RR for the

NS RRs, one SIG RRs for each name server's A RR, and one SIG RRs for each name server's KEY RR. These 30 SIG RRs cost about 53ms to verify. Considering the fact that the NS query is one of the most frequently used query types, we propose the addition of a special SIG RR that covers all of these NS RRs, A RRs, and KEY RRs in an NS query answer to minimize verification time. Figure 6-4 shows that the proposed special SIG RR dramatically reduces the *elapsed time* of a NS query in BFT-DNS-2. The size of the reply packet also decreases, as does the *server time*.

Test System	**#RR/ #SIG**	**Reply size(byte)**	**Elapsed Time (ms)**	**Server Time (ms)**	**Server CPU**
normal	23/10	1567	59.9	1.05	2.42%
has a special SIG RR	15/2	883	15.8	0.788	7.15%

Figure 6-4: Use BFT-DNS-2 to Query Name Servers of a Zone

6.4.3 Update Request

Figure 6-5 shows the results of the tests of A RR update requests. We only tested these requests with BFT-DNS-1. The update requests in BFT-DNS-2 will be discussed in Section 6.5. TIS/DNS and TIS/DNSSEC do not support update transactions.

Test System	**Elapsed Time (ms)**	**Server Time (ms)**	**Server CPU**
BFT-DNS-1	4.11	2.35	45.3%

Figure 6-5: Update the IP Address of a Host

The *elapsed time* and the *server time* of an update request both increase by about 2ms, compared to a read-only request. This is because of an extra message roundtrip and extra cryptographic operations introduced by the replication library for update requests. The overhead introduced by checkpoint generation and garbage collection is very small and can be ignored.

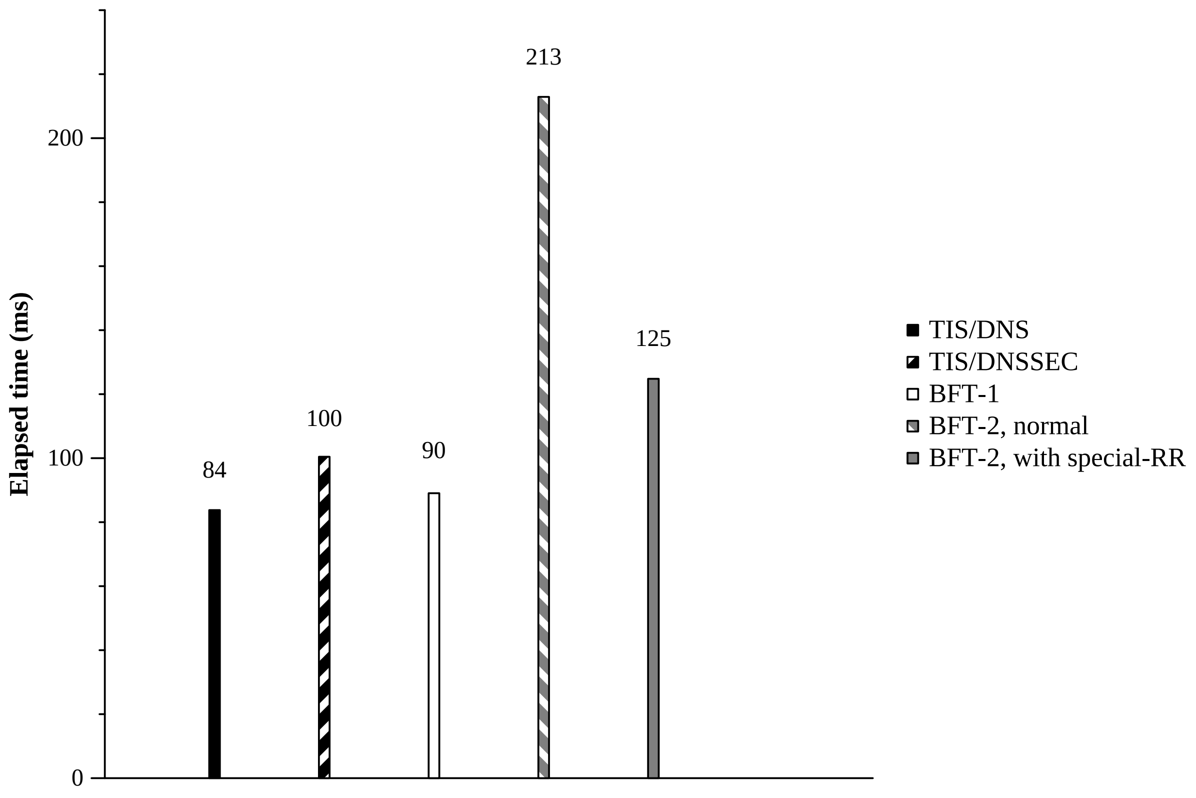


Figure 6-6: Estimated Total Elapsed Time of an A RR Query That Involves Name Servers of 3 Different Zones

6.4.4 System Performance Comparison

Client Elapsed Time

The client *elapsed time* is one of the most important metrics in the DNS performance evaluation. It determines how fast the client can find the information it needs. Figure 6-6 shows the estimated client elapsed time for an A RR request that needs to contact name servers of 3 different zones. A query for IP address of `www.netscape.com` from `chord.lcs.mit.edu` is an example of this kind of requests. The total client *elapsed time* includes the *elapsed time* of two NS RRs queries and the *elapsed time* of one A RR query. We use the test results presented in the above sections to get the estimated values. We also assume that the network round trip delay is 20ms, which is an optimistic value in the Internet today, so we add 60ms to each client total *elapsed time*.

In TIS/DNS, TIS/DNSSEC and BFT-DNS-1, the network round trips delay is the dominant part of the *elapsed time*. So their *elapsed times* are very close. The *elapsed times* in our BFT-DNS-1 is 11% less than in TIS/DNSSEC and only 6% greater than in TIS/DNS. The normal BFT-DNS-2 is 110% slower than the TIS/DNSSEC, but a 210ms elapsed time is still acceptable in most cases. The BFT-DNS-2 with our proposed new special SIG RR is only 24% slower than TIS/DNSSEC.

Server Throughput

Server throughput is another one of the most important metrics in the DNS performance evaluation, especially in frequently visited zones.

We use the following equations to estimate the server throughput of each system.

$$throughput_{server} = \frac{1}{time_{elapsed} \times CPU_{server}}$$

Since there is no disk operation in our tests and the network is not the bottleneck, this estimation is reasonable. The results are presented in Figure 6-7 and Figure 6-8.

Test System	**A RR Query**	**NS RR Query**	**Update Request**
TIS/DNS	2539	1922	N/A
TIS/DNSSEC	1885	1111	N/A
BFT-DNS-1	2311	1064	537
BFT-DNS-2	1500	690[a]/883[b]	$\ll 1$ [c]

[a]normal
[b]has a special SIG RR
[c]depended on the size of zone database

Figure 6-7: Estimated Maximum Server Throughput (requests/sec)

First, we analyze and compare the A RR request throughputs. BFT-DNS-1 is 23% faster than TIS/DNSSEC and is only 9% slower than TIS/DNS. This is a very impressive result. BFT-DNS-2 is 21% slower than TIS/DNSSEC, but it does offer a more secure service.

We then analyze and compare the NS RR request throughputs. BFT-DNS-1 is only 4% slower than TIS/DNSSEC and 45% slower than TIS/DNS. BFT-DNS-2 is 21% slower than TIS/DNSSEC. Again, considering the more secure service BFT-DNS offered, the performance degradation is acceptable.

BFT-DNS-1 can handle 537 update requests per second. This should be far more than enough in today's Internet. We will discuss the update performance of BFT-DNS-2 in the next section.

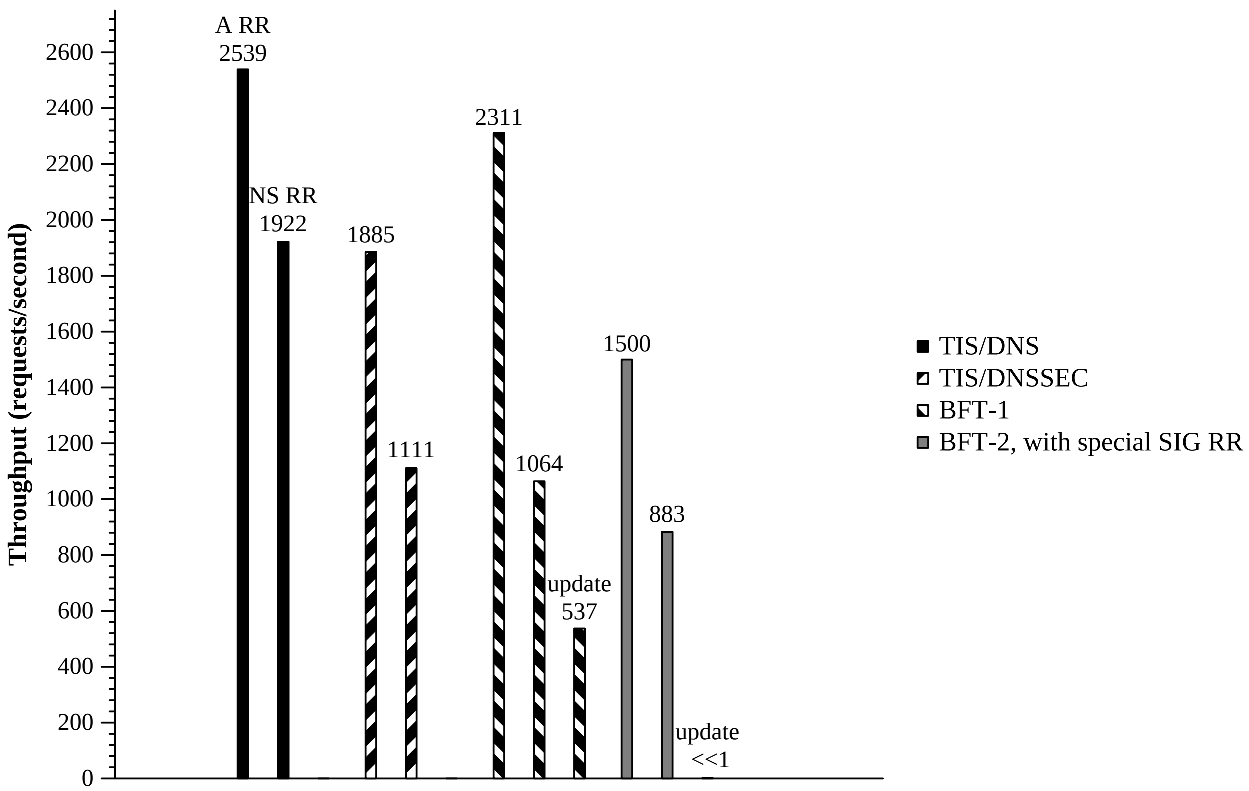


Figure 6-8: Extimated Max. Server Throughput

6.5 Performance Issues

6.5.1 Cost to Establish a Session Key in BFT-DNS-1

We denote the BFT-DNS scheme 1 with session key cached as BFT-DNS-1-K, the BFT-DNS scheme 1 without session key cached as BFT-DNS-1-NK in this subsection.

Figure 6-9 compares the test results of BFT-DNS-1-NK and BFT-DNS-1-K. Although data shows that establishing a session key is quite expensive, it should not degrade the overall performance of BFT-DNS scheme 1 significantly, since this case should be rare if the session key cache mechanism works well (see Section 5.4.6, 7.2, and 7.3).

Operation	**Session key in Cache**	**Elapsed Time (ms)**	**Server Time (ms)**	**Server CPU**
IP RR Query	no	145.9	136.7	93.7%
IP RR Query	yes	2.18	0.205	19.8%
IP RR Update	no	280.2	139.2	49.0%
IP RR Update	yes	3.45	0.673	27.3%

Figure 6-9: Comparison of BFT-DNS-1-NK and BFT-DNS-1-K

We analyze the test result of IP RR query first. The *elapsed time* and *server time* in BFT-DNS-1-NK are almost 2 orders of magnitude larger than in BFT-DNS-1-K. The dominant part is the procedure of setting up a session key, which involves PPK operations. PPK operations are very expensive. We conducted a simple test of the speed of these PPK operations. In this test a 128-bit session key, which is also used in BFT-DNS, was encrypted, decrypted, signed, and verified 10000 times using RSA PPK algorithm. The modulus length and public exponent of the RSA keys used in this test are the same as in BFT-DNS (i.e. 512 bits and 3). Figure 6-10 shows the result. In the process of setting up a session key for read purpose, a client needs to do $3f + 1 = 4$ encrypt operations and a server needs to do one decrypt operation, which is 100 times slower than an encrypt operation. So the server side is the bottleneck of this process.

We then analyze the test results for IP RR update requests. The *elapsed time* of an IP RR update request in BFT-DNS-1-NK is almost doubled as compared to a read-only request (IP RR query). This is because the client must provide the proper signature to prove its authenticity and authority in the process of setting

Operation	Time(ms)
Encrypt	1.63
Decrypt	129.4
Sign	129.5
Verify	1.55

Figure 6-10: RSA PPK Operation Time of a 32-byte Message

up an update session key. The generation of a signature costs about 130ms in our tests. The *server time* increases about 3ms because of the extra message roundtrip and extra cryptographic operations introduced by the replication library for update requests. In addition, the server needs to verify the signature of the client, which costs about 1.5ms. BFT-DNS-1 can handle 7.3 update requests per second even without the session key cached. This should satisfy most zones. Also remember that our test machines are at least 10 times slower than the current commercially available top class servers.

For any type of request, BFT-DNS-1-NK is at least 95% slower than the BFT-DNS-1-K. As we have mentioned, this is due to the high cost of setting up a session key. As a result, we should reduce the session key miss rate as much as possible.

RSA is not the only good PPK algorithm. Rabin and Williams' algorithm [Rab79, Wil80] is at least as secure as RSA, and is much faster. Using the same machine and same key length (512 bits) as our other tests, test data of Rabin and Williams' algorithm (thanks to M. Castro) shows an encrypting operation takes about 31ms and a decrypting operation takes 0.3ms. So Rabin and Williams' algorithm is 4 times faster than the RSA library we used in our BFT-DNS implementation. If we use Rabin and Williams' algorithm to establish sessions, a substantial increase in server throughput and a substantial decrease in *elapse time* in BFT-DNS-NK are foreseeable.

6.5.2 SOA SIG RR Refreshing Interval in BFT-DNS-2

In all of our BFT-DNS-2 tests, we disabled the SOA SIR RR update request generator to get data for *Server CPU*. The *elapsed time* should not be affected because the SOA SIG RRs could be generated in a dedicated thread with low priority running in the background. Actually a SOA SIG RR could be generated during server idle time before the corresponding SOA SIG RR update request comes in. This will not cause any security problems.

Our tests show that if the SOA SIG RR update interval is 10 seconds, the *Server CPU* usage percentage will increase by about 1.4, so the server throughput will decrease by about 1.4%. If we increase the interval or use faster machines, this overhead becomes even lower. In some zones that demand more security, we can make the interval shorter. If the interval is 1 second, the overhead will be 7%, which is still small. Making the interval shorter than 1 second may not help a lot to provide a more secure service, since the security level is determined by the TTL of a SOA SIG RR, which is bounded by the following equation.

$$TTL_{SIG_{soa}} \geq Interval_{sign} + Delay_{network} + Discrepance_{clocks}$$

The value of network delay can be as big as several hundred milliseconds in the Internet. Furthermore, it is very difficult to make the clock discrepancy between every pair of hosts in the Internet be very small.

6.5.3 Update in BFT-DNS-2

In BFT-DNS-2, once an update happens, replicas are required to re-generate every SIG RR in their databases. In our tests, 42 SIG RRs are re-generated, which takes 5.82 seconds. It dominates both the client *elapsed time* and the *server time*. This will be worse in bigger zones. A zone that contains 1000 SIG RRs will need more than 2 minutes to do a single update. As a result, BFT-DNS scheme 2 is only practical in small and relatively stable zones.

6.6 Comparison of BFT-DNS Scheme 1 and BFT-DNS Scheme 2

We show a comparison of BFT-DNS scheme 1 and BFT-DNS scheme 2 below, based on the performance data we got in the above sections.

- scheme 1:

  pro
  - Read request throughput is very high, if the session key is in the caches.
  - Update operation is fast.

  con
  - Servers need to cache huge number of session keys.

    - If the session key is not in the caches, it is expensive to set up one.

- scheme 2:

  pro
  - Read request throughput is always high.
  - Need no session key and no cache for session keys.

  con
  - Update operation is expensive and slow, especially in large zones.
  - Not as secure as scheme 1. Still has a small freshness attack window.

Apparently, scheme 1 should be used in the circumstance in which the session key miss rate can be kept low, while scheme 2 should be used in some small and relatively stable zones in which caching session keys is not practical. We will discuss this further in Chapter 7.

Chapter 7

Discussion

7.1 Overview

In the previous chapters we discussed an iterative approach to implement BFT-DNS operations; that is, a resolver sends a request to name servers of a zone and gets the answer without that name servers contacting name servers of other zones. In this chapter, we will discuss how to build a Byzantine-fault-tolerant DNS Infrastructure based on this approach. Our main problem is how a query request traverses the domain name space tree to get the final answer. It seems the BFT-DNS scheme 1 is better than scheme 2 (see Section 6.6) if the session key cache mechanism works well. So we will focus on how to make the session key cache works efficiently while being affordable.

We will first propose two different approaches to query resolution. Since all update requests in DNS involve only their target zones, these approaches are only concern read requests, which may involve several zones. Then we will discuss the RR cache mechanism.

7.2 Query Resolution Approach 1: Partially Recursive

In this approach, queries are resolved using a hybrid of recursive resolution and iterative resolution. (See Section 2.5) Most of the burden of resolution of a query is put on the name servers of a resolver's local zone. Resolvers only contact local name servers and send recursive queries to them. Local name servers act as *query proxies*. They

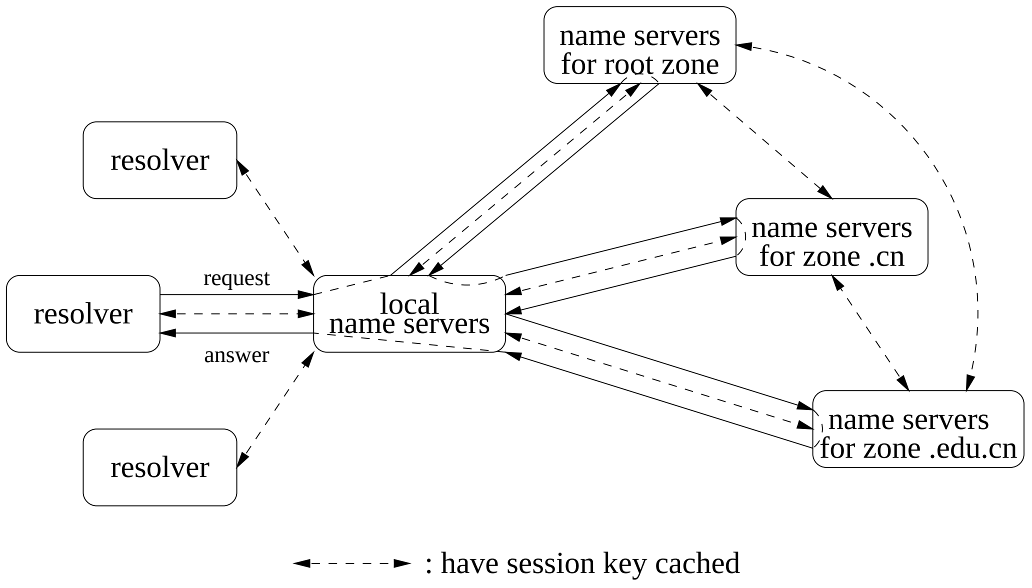

name servers for root zone
resolver
name servers for zone .cn
request
resolver
local name servers
answer
name servers for zone .edu.cn
resolver
: have session key cached

Figure 7-1: Query Resolution Approach 1: Partially Recursive

handle these queries using iterative resolution, that is, contact name servers of the proper zones one by one until they find the answer. See Figure 7-1 for an example.

In this approach, most of zones are using BFT-DNS scheme 1. To take advantage of BFT-DNS scheme 1, we should use cached session keys as much as possible. Session keys between resolvers and their *query proxies* are easy to cache because the number of resolvers served by the local name servers is relatively small. The main problem of this approach is that each name server that uses BFT-DNS scheme 1 may need to cache the session keys for all the *query proxies* in the Internet.

According to [Rut99], there are 45 million hosts in the Internet. If we assume every 100 hosts share one set of local name servers, or *query proxies*, there are about 450,000 sets of *query proxies*. So a server that uses BFT-DNS scheme 1 need to cache up to $450,000 \times 4 = 2,000,000$ session keys. Suppose each session key consumes 25 bytes; then we need 50Mbytes to cache all the session keys. 50 Mbytes of memory should not be a big deal for name servers of a large or moderate size zone, since it is reasonable to assume the organization that owns such zone has enough resources to afford this kind of server. We also expect that most small zones' owners can afford this kind of server. But in some case, this requirement may be still too high. Session keys can be cached on disks, but it may substantially reduce the performance. In addition, session keys should not be valid forever. If the average TTL of a session key is 20 days, on average one session key expires in the 2,000,000 session keys every second. Thus, a name server need to set up one new session keys per second. The burst rate is even higher. This will add a significant workload to some old servers such as those we used in our experiments. Fortunately, small zones are supposed to be very stable. In such case, our BFT-DNS scheme 2, which has little requirement on the memory and computation ability of servers, is more suited for these zones.

7.3 Query Resolution Approach 2: Completely Recursive

In this approach, queries are handled recursively. Whenever a name server receives a recursive query and finds out the query is beyond the range of its administrated zone, it sends a recursive query to the "closest known" servers (See Section 2.5) to oblige them to find the answer. This process continues until the final answer is found. See Figure 7-2 for an example.

In this approach, all zones can use BFT-DNS scheme 1, since every name server only needs to cache session keys for its superzone and subzone. Name servers for top zones still need to cache huge number of session keys. For example, the *.com* zone has about 2 million subzones [Int99]. Although many of its subzones share name servers with others, the total number of the session keys that need to be cached in the name servers of *.com* zone is still a huge number. We assume the organization that owns such a top level zone has enough resource to afford sufficiently powerful servers. Small zones only need to cache a few session keys, since the number of their superzones is small and they generally have no or just a few subzones.

The biggest disadvantage of this approach is that the workload of the top zones, especially the root zone, increases substantially. Almost all non-local queries will reach the root zone. The name servers of the root zone cannot just refer the querier to a different name server, but are instead obliged to send the query to other name servers to get the answer and send back the answer.

7.4 RR Caching

Caching is a very important property in today's DNS. It offloads the root servers and reduces the DNS traffic on the Internet. In our BFT-DNS, caching is not allowed if users require the freshest data, since the cached data may be stale before its TTL

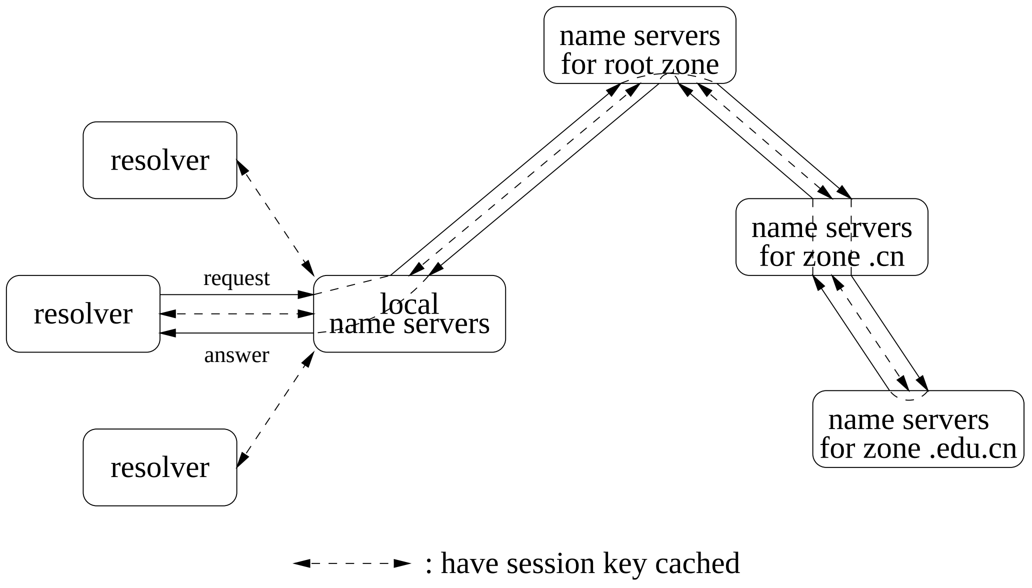

name servers for root zone
resolver
resolver
resolver
request
answer
local name servers
name servers for zone .cn
name servers for zone .edu.cn
: have session key cached

Figure 7-2: Query Resolution Approach 2: Completely Recursive

expires. But abandoning the caching mechanism may cause the root servers to be overloaded, since all non-local DNS requests in the whole Internet will be sent to the root servers in both our resolution approaches. To solve this problem, we can allow some caching. We can allow name servers to cache RRs received in recent queries. Some zones can make some sensitive RRs in its database non-cacheable, that is, set their TTL to zero. Also users can choose if they want to take advantage of caches to get quick service or disable the cache to get the most accurate information. In many cases, the most secure services is not required; for example, a user may not take care very much about security issues when he/she visits http://www.disney.com unless he/she wants to buy something online from this site. On the other hand, users can ask for the freshest public key of another user to verify an important signature. In this case, the user can ask the resolver to send a query with a special bit set to notify the name servers to bypass their caches.

Caching NS RRs has some other important benefits. If name servers find the NS RRs that contains the name servers of the target zone of a query in their RR caches, and the corresponding session keys are found in their session key caches, they can get the answer for the query from the target name servers via a "short path". That is, they can contact the target name servers directly without passing the query to the top zone. The session keys are used to verify the identities of the target name servers, so even a stale NS RR will not fool the name servers. They can just abort the "short path" and do the query resolution in the normal way (without using cached NS RRs). This optimization should significantly decrease the workload in top zones.

Chapter 8

Conclusions

In this thesis, we have analyzed the security problems with DNS and the new proposed DNS security extension, and we have presented the design, implementation and performance evaluation of a Byzantine-fault-tolerant DNS, or BFT-DNS. This chapter provides a summary of the thesis. It also suggests directions for future work.

8.1 Summary

Chapter 2 describes what DNS is and how it works. DNS is a distributed database used to provide important information about domains or hosts in the Internet. The name space and administration of DNS are both hierarchical. The administration unit of DNS is *zone*. A zone database consists of data entries called DNS *resource records*, or RRs. Each RR holds a type of data for a domain name. DNS has two types of active components: name servers and resolvers. A resolver sends a request to a name server to ask for one or a set of RRs. Resolvers and name servers interact each other to distribute the public information of a domain or a host to everyone who need it. There are 2 ways to resolve a query: recursive and iterative.

Chapter 3 analyses some security problems with DNS. DNS was not designed to be a secure protocol. It does not provide any way for client to verify the authenticity of the data returned by a DNS query. Meanwhile, DNS plays a key role in directing the information flow in the Internet. Furthermore, the RRs queried from DNS sometimes are even used for checking authentication in other Internet protocols. As a result, DNS is a favorite target of hackers. There are several ways to attack DNS, which can cause serious damages. Furthermore, it is almost impossible to do dynamic update in DNS. A DNS security extension has been proposed to fix the DNS security flaws. In this extension, authentication is provided by associating with RRs in the DNS cryptographically generated digital signatures. Every zone has a key to generate these digital signatures, or SIG RRs. A resolver can verify the final answer it got thought an authentication chain. Dynamic update is also possible in this DNS security extension. Although the DNS security extension is more robust than the original DNS, it is not secure enough. There are several security flaws in the system. If the zone key is kept online, a single name server break-in may still subvert the whole zone. If the zone key is kept off-line, the dynamic update data are no longer protected. In both case, the DNS security extension is vulnerable to freshness attacks. In addition, simple replica scheme is used in zone transfer, causing consistency and reliability problems.

Chapter 4 describes how to use CLBFT, a practical Byzantine-fault-tolerant algorithm, to design a Byzantine-fault-tolerant DNS. To build a practical Byzantine-fault-tolerant DNS with data consistency, and high reliability and availability, we use the newly invented CLBFT algorithm. CLBFT is the only practical Byzantine-fault-tolerant algorithm available today that can work in an asynchronous network like the Internet. In our new designed BFT-DNS, each zone has $3f + 1$ tightly coupled name servers to survive up to f Byzantine faults. A client and name servers of a zone communicate with each other using a protocol based on CLBFT to process a query or an update request. Some optimizations are done to improve the performance of read only queries. We designed two read schemes. The first one uses the message authentication code to authenticate the reply data. The second one uses pre-generated SIG RRs to authenticate the reply data. In addition, a recently generated SOA SIG RR with very short TTL is returned to the client to indicate the reply is *almost* up-to-date. BFT-DNS can support update requests and dynamically updated data are as safe as other data. There is one public-private key pair for every name server. The private keys are stored online, but a hacker still needs to break in at least $f + 1$ name servers of a zone to do any damage.

Chapter 5 discusses the implementation our BFT-DNS. Our BFT-DNS implementation is based on *TIS/DNSSEC*, a beta version of DNS implementation with secure DNS extensions developed by Trusted Information Systems, Inc. The BFT Replication Library developed by M. Castro and B. Liskov plays a key role in our implementation. Simply speaking, we built the BFT-DNS by combining TIS/DNSSEC and the BFT Replication Library. We also added some new modules, such as the update request authorization module and the checkpoint manager module, to the system.

Chapter 6 presents the performance evaluation of BFT-DNS. We compared BFT-DNS with TIS/DNSSEC and DNS without security feature. We shows that our BFT-DNS scheme 1 performs as well as or even better than TIS/DNSSEC if the session key is cached, but is slow when the session key is not in the cache. BFT-DNS scheme 2 performs slightly slower than TIS/DNSSEC, but it is still acceptable. Update operations in BFT-DNS scheme 1 are fast enough for most zones, while in BFT-DNS scheme 2, update operations are only practical in small and relatively stable zones.

Chapter 7 discusses some extensions to BFT-DNS to overcome its limitations and improve its overall performance. Session key caching mechanism is essential to the BFT-DNS scheme 1. We propose two resolution approaches, a partially recursive one and a completely recursive one, to make the session key caching mechanism of BFT-DNS scheme 1 more feasible and more efficient. We also discuss the RR caching mechanism in BFT-DNS, which will decrease the workload in the top zone name servers substantially.

8.2 Future Work

8.2.1 Improve the Implementation

Our implementation of BFT-DNS is still a test system. To build a usable and fully functional BFT-DNS software package, much works is left to be done.

We have not implemented the view change and failure recovery in our BFT-DNS, This will affect the liveness of our system. We also have not implemented packet re-transmission, as discussed in 5.5.

The two resolution approaches and the RR caching mechanism, which are described in Chapter 7, have not been implemented yet. In addition, further research on DNS service access patterns in various types of zones is needed to design some better resolution approaches and better session key caching policies, which are essential to the overall performance of the BFT-DNS.

In our current implementation, we do not use multiple threads or multiple processes. The incoming requests are serialized and are served one by one. If one or several requests take too much time to process, such as an update request in scheme

2, other requests in the queue may wait too long or even time out. If we use multiple threads, this problem will be solved. Some concurrency control mechanisms need to be added as well.

8.2.2 Compatibility

Our current BFT-DNS implementation is not compatible with the DNS we are using and the DNS security extension for the following reason.

1. Additional information for the CLBFT protocol is added to request and reply messages in BFT-DNS.

2. Resolvers need $2f + 1$ replies from different replicas to get a convincing answer in BFT-DNS.

3. SIG RRs in BFT-DNS scheme 2 are slightly different from the SIG RRs in DNS security extension.

Making BFT-DNS compatible with the original DNS is very important, or BFT-DNS will never be used in the real world. There are two key points in achieving this compatibility.

1. A BFT-DNS server must be able to handle the requests from old resolvers correctly, and must send back recognizable replies.

2. A BFT-DNS resolver must be able to know whether the zone it will contact is a BFT-DNS. If not, the resolver must be able to send a request recognizable by the old name servers. So we propose to add some additional information to NS RRs to indicate the type of a zone.

8.2.3 Privacy Issues

In BFT-DNS we did not pay attention to privacy issues, since it is part of the design philosophy of the DNS that the data in it are public and that the DNS gives the same answers to all inquirers. In some circumstance, zone data may be required to be available to a group of designed users, for example, some internal public keys of a company should be available only to employees of the company. Simply adding some read request authentication and access control is not enough. A compromised replica may leak information to an attacker. So an extension to the CLBFT algorithm, combined with a secret-sharing algorithm [Kot95, Sha79, Bla79], is required to provide Byzantine-fault-tolerant privacy.

Bibliography

[AK96] Ross Anderson and Markus Kuhn. Tamper resistance - a cautionary note. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, pages 1–11, Oakland, CA, November 1996.

[AL98] Paul Albitz and Cricket Liu. *DNS and BIND, Third Edition.* O'Reilly and Associates, Inc., 1998.

[Bla79] G. R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the National Computer Conference v.48*, pages 242–268, 1979.

[BM97] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing:incrementality at reduced cost. In *Advances in Cryptology - Eurocrypy 97*, 1997.

[CL99a] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI99)*, New Orleans, USA, February 1999.

[CL99b] M. Castro and B. Liskov. Authenticated byzantine fault tolerance without public-key cryptography. Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, 1999.

[CL99c] M. Castro and B. Liskov. Practical recovery in a byzantine fault-tolerant-algorithm. In *Submission for publication*, May 1999.

[CR92] R. Canneti and T. Rabin. Optimal asynchronous byzantine agreement. Technical Report 92-15, Computer Science Department, Hebrew University, 1992.

[Eas97] D. Eastlake. Secure domain name system dynamic update. Technical Report DARPA-Internet RFC 2137, CyberCash Inc., April 1997.

[Eas99a] D. Eastlake. Domain name system security extensions. Technical Report DARPA-Internet RFC 2535, IBM, March 1999.

[Eas99b] D. Eastlake. Dsa keys and sigs in the domain name system. Technical Report DARPA-Internet RFC 2536, IBM, March 1999.

[Eas99c] D. Eastlake. Rsa/md5 keys and sigs in the domain name system. Technical Report DARPA-Internet RFC 2537, IBM, March 1999.

[EK97] D. Eastlake and C. Kaufman. Domain name system security extensions. Technical Report DARPA-Internet RFC 2065, CyberCash, Iris, January 1997.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. In *Journal of the ACM 32(2)*, 1985.

[GM98] J. Garay and Y. Moses. Fully polynomial byzantine agreement for n ¿ 3t processors in t+1 rounds. In *SIAM Journal of Computing 27(1)*, 1998.

[HW87] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of 14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[Int99] Network solutions, inc, 1999. Available at http://www.networksolutions.com.

[KMMS98] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The securering protocols for securing group communication. In *Hawaii International Conference on System Sciences*, 1998.

[Kot95] S. C. Kothari. Generalized linerar thresh old scheme. In *Advances in Cryptology: proceedings of CRYPTO 84*, pages 231–241, 1995.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine gererals problem. In *ACM Transactions on Programming Languages and System, 4(3)*, 1982.

[MD88] Paul V. Mockapetris and Kevin J. Dunlap. Development of the domain name system. 1988.

[Moc87a] P. Mockapetris. Domain names - concepts and facilities. Technical Report DARPA-Internet RFC 1034, ISI, Nov 1987.

[Moc87b] P. Mockapetris. Domain names - implementation and specification. Technical Report DARPA-Internet RFC 1035, ISI, Nov 1987.

[Rab79] M. O. Rabin. Digital signatures and public-key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, MIT Lab. for Computer Science, Cambridge, MA, January 1979.

[Rei96] M. Reiter. A secure group membership protocol. In *IEEE Transactions on Software Engineering 22(1)*, 1996.

[Riv92] R. Rivest. The MD5 message-digest algorithm. Internet RFC-1321, april 1992.

[RSA78] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM, v. 21, n. 2*, pages 120–126, February 1978.

[RSA79] R. L. Rivest, A. Shamir, and L. M. Adleman. On digital signatures and public key cryptosystems. Technical Report MIT/LCS/TR-212, MIT Lab. for Computer Science, Cambridge, MA, January 1979.

[Rut99] Anthony M. Rutkowski. Internet trends, 1999. Available at http://www.mids.org/mapsale/data/trends.

[Sch90] F. Schneider. Implementing fault tolerant service using the state machine approach : A tutorial. In *ACM Computing Surveys, 22(4)*, pages 299–319, December 1990.

[Sha79] A. Shamir. How to share a secret. In *Communications of the ACM, v.24, n.11*, pages 612–613, November 1979.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1.* Addison Wesley Longman, Inc., 1994.

[TIS99] Network association. inc secure dns product, 1999. Available at http://www.nai.com/products/security/tis_research/netsec/net_dns.asp.

[Tsu98] G. Tsudik. Message authentication with one-way hash functions. In *ACM Computer Communications Review, 22(5)*, 1998.

[VTRB97] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. Dynamic updates in the domain name system (dns update). Technical Report DARPA-Internet RFC 2136, ISC, Bellcore, Cisco, DEC, April 1997.

[Wil80] H. C. Williams. A modification of the rsa public-key encrption procedure. In *IEEE Transactions on Information Theory v.IT-26, n. 6*, pages 726–729, November 1980.