An Intrusion Tolerance Approach for Protecting Network Infrastructures

by

Steven Cheung

B.S. (University of Hong Kong) 1989
M.Phil. (University of Hong Kong) 1992
M.S. (University of California, Davis) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at DAVIS

Committee in charge:

Professor Karl N. Levitt, Chair
Professor Matthew A. Bishop
Professor Biswanath Mukherjee

1999

The dissertation of Steven Cheung is approved:

Chair Date

Date

Date

University of California at Davis

1999

An Intrusion Tolerance Approach for Protecting Network Infrastructures

Copyright 1999
by
Steven Cheung

Contents

List of Figures vi

List of Tables vii

1 Introduction 1

2 An Efficient Message Authentication Scheme for Link State Routing 6
- 2.1 Introduction . . . 6
- 2.2 Background: Link State Update Authentication . . . 8
- 2.3 Optimistic Link State Verification . . . 11
  - 2.3.1 Assumptions . . . 11
  - 2.3.2 Protocol Overview . . . 13
  - 2.3.3 Sender Process . . . 14
  - 2.3.4 Receiver Process . . . 16
  - 2.3.5 Recovery Process . . . 17
  - 2.3.6 An Example . . . 18
  - 2.3.7 Cost Analysis . . . 20
- 2.4 Discussion . . . 21

3 Protecting Routing Infrastructures from Denial of Service 24
- 3.1 Introduction . . . 24
- 3.2 Examples of Routing Infrastructure Failures . . . 26
- 3.3 Related Work . . . 28
- 3.4 Our Model . . . 29
- 3.5 Our Approach . . . 32
- 3.6 Distributed Probing . . . 33
- 3.7 Flow Analysis . . . 38
- 3.8 Discussion . . . 43

4 Protecting Domain Name Systems 47
- 4.1 Introduction . . . 47
- 4.2 A Motivating Example . . . 48
- 4.3 Overview of DNS . . . 49
  - 4.3.1 What is DNS? . . . 49

4.3.2 How does DNS Work? 50
4.3.3 DNS Message Format 51
4.4 DNS Vulnerabilities 52
4.5 Related Work 54
4.5.1 Hardening BIND—Vixie's Approach 55
4.5.2 Dnsproxy 58
4.5.3 DNS Security Extensions 60
4.5.4 Evaluation 61
4.6 Our Approach 68
4.7 System Model 69
4.7.1 DNS Data 70
4.7.2 Trust, Authority, and Delegation 73
4.7.3 Resolvers 73
4.7.4 Name Servers 75
4.7.5 Assumptions 79
4.7.6 Our Goal 80
4.8 Our DNS Wrapper 81
4.8.1 Wrapper-based Design Versus Sniffer-based Design 82
4.8.2 Overview of the Specification of Our DNS Wrapper 84
4.8.3 The Specification of Operation $Wrapper_sq$ 85
4.8.4 The Specification of Operation $Wrapper_sr$ 86
4.8.5 The Specification of Operation $Wrapper_sr1$ 87
4.8.6 The Specification of Operation $Wrapper_sr2$ 88
4.8.7 The Specification of Operation *AuthVerified* 89
4.8.8 The Specification of Operation *CheckAuthServer* 91
4.9 Experiments 93
4.9.1 Overview 93
4.9.2 General Experimental Setup 93
4.9.3 Experiment #1 94
4.9.4 Experiment #2 99
4.9.5 Experiment #3 101
4.10 Discussion 104
4.11 Summary and Future Work 105

5 Conclusions and Future Work **107**
5.1 Conclusions 107
5.1.1 Intrusion Tolerance: A New Approach 108
5.1.2 On Efficient Message Authentication for Link State Routing 110
5.1.3 On Protecting Routing Infrastructures from Denial of Service 111
5.1.4 On Protecting Domain Name Systems 112
5.2 Future Work 113
5.2.1 On Optimistic Link State Verification 113
5.2.2 On Protecting Routing Infrastructures from Denial of Service 115
5.2.3 On Protecting Domain Name Systems 116

Bibliography **117**

A BIND 4.9.5 Resource Record Filtering Algorithm **124**

B Additional Specifications for Our Wrapper **126**

C Top 100 Web Sites **130**

List of Figures

2.1 The Sender Process. 15
2.2 A Network and an Associated BRUP Graph. 19

3.1 Black Hole Routers and Misrouting Routers. 27
3.2 Classifications of Bad Routers. 31
3.3 Testable Edges. 34
3.4 Conservation of Transit Traffic: $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$. 39
3.5 $t_{(i,j)}(k)$, $n_{(i,j)}(k)$, and $g_{(i,j)}(k)$, $k \in \{i, j\}$. 40

4.1 Hierarchical Structure of DNS Name Space 49
4.2 Iterative Name Resolution 51
4.3 Example query. 57
4.4 Erroneous Response #1. 57
4.5 Erroneous Response #2. 58
4.6 Erroneous Response #3. 59
4.7 Erroneous Response #4. 63
4.8 A Security Log Message for www.zone.com. 97
4.9 A Security Log Message for www.3com.com. 98

List of Tables

4.1 Summary of Evaluation. 67
4.2 Cumulative Response Time (in Sec.) for the "Top 100 Web Sites" Data Set. 95
4.3 Cumulative Response Time (in Sec.) for the "Two-day trace" Data Set. . . 100
4.4 System and User Times Used (in Sec.) for the "Two-day Trace" Data Set. . 101

Acknowledgements

I am grateful to Karl Levitt, my thesis adviser, for his guidance on my research, for his constant encouragement, and for his teaching me how to survive professionally—in particular, writing grant proposals and doing presentations.

I would like to thank Matt Bishop and Biswanath Mukherjee, my dissertation committee members, for their invaluable comments that improve the presentation of this dissertation. I would also like to thank Christina Chung and Nick Puketza for proofreading an earlier draft of this dissertation and for their useful suggestions.

This work was supported by the Defense Advanced Research Projects Agency under grant ARMY/DAAH 04-96-1-0207.

It has been a great pleasure to study in the Computer Science Department of UC Davis. Through lectures, seminars, and discussions, I have gained exposure to various areas in computer science and received helpful feedbacks on my research. I thank everyone in the Computer Science Department for their help and friendships.

Many thanks to the colleagues with whom I have worked on research projects and papers: Matt Bishop, Kirk Bradley, Rick Crawford, Mark Dilger, Jeremy Frank, Jim Hoagland, Calvin Ko, Francis Lau, Karl Levitt, Biswanath Mukherjee, David O'Brien, Ron Olsson, Nick Puketza, Jeff Rowe, Steven Samorodin, Stuart Staniford-Chen, Stuart Stubblebine, Steven Templeton, Chris Wee, Raymond Yip, and Dan Zerkle. I have learned a great deal from them—technical and otherwise.

I am indebted to my family for their love and support. I dedicate this dissertation to them.

Chapter 1

Introduction

Through a myriad of applications, including electronic mail, WWW, and electronic commerce, computer networks play an increasingly important role in many aspects of our lives. Security incidents like the Melissa macro virus [11], the "smurf" ICMP[1] denial of service attacks [9], the IP spoofing and TCP connection hijacking [8], and the Morris' Internet worm [65], demonstrate how vulnerable current computer systems and networks are to attacks. The costs of security breaches can be very high: unauthorized information disclosure, loss of data integrity, and system degradation or unavailability. Thus network security becomes crucial.

Most of the existing network security work concerns confidentiality, data integrity, user authentication, and non-repudiation all on an *end-to-end* basis. In contrast, protecting the underlying network infrastructures has received little attention until recently—when we have become more aware of how vulnerable our existing network infrastructures are to attacks and of their severe impacts. End-to-end security can be foiled by attacks that exploit vulnerabilities in the network infrastructures. For example, a compromised router can drop packets to cause denial of service. In this dissertation, we describe our work on protecting two core components of network infrastructures—routers and domain name systems—both of which are security critical.

Attacks on routing infrastructures can be classified as follows: packet generation (e.g., masquerading as a certain host or router to send forged packets), packet alteration (e.g., modifying link state information of routing control packets or modifying data packets

[1] ICMP stands for "Internet Control Message Protocol", an integral part of the Internet Protocol (IP) that handles error and control messages.

in transit), packet removal (e.g., dropping routing control packets or data packets to cause denial of service), misrouting (e.g., routing packets in the "wrong" direction so that they will take longer or forever to reach their destinations), and breach of confidentiality (e.g., performing traffic analyses or eavesdropping on data packets).

The use of tools like Secure Sockets Layer (SSL), Pretty Good Privacy (PGP), and Secure Shell (SSH) has become widespread for protecting the confidentiality and integrity of data packets. For example, web browsers are equipped with SSL to enable secure information transmission (e.g., sending credit card numbers) from a machine to another over an insecure network. However, using encryption and message authentication schemes on an end-to-end basis cannot prevent data packets from being removed or misrouted in the network. To fill this gap, we present protocols that detect and respond to misbehaving routers to protect packets from denial of service attacks.

Routers exchange control packets to inform other routers about their status (e.g., up/down link states and link costs[2]). Based on the routing control packets received, a router computes a routing table that is used to forward incoming packets toward their destinations. Usually there is no need to protect the secrecy of routing control packets. However, protecting the integrity and the authenticity of these control packets is security critical. If an attacker can successfully forge or modify routing control packets, the routers that use the incorrect information in those control packets may route packets incorrectly. As a result, packets may suffer from long delays or may not reach their destinations. To protect the integrity and the authenticity of routing control packets, message authentication schemes have been proposed for routing protocols. Previous work either is very expensive computationally (e.g., public-key based message authentication schemes) or has certain limitations (e.g., the maximum clock skew among routers must be bounded by a specified threshold). We present an efficient message authentication scheme for link state routing that does not have these limitations.

In a domain name system (DNS), distributed name servers collaborate to provide name service (e.g., mapping host names to IP addresses). Many network applications, such as file transfer, remote login, WWW, and electronic mail, depend on DNS in a security related fashion. For example, if an attacker can cause a client to use incorrect DNS data,

[2]There are different cost metrics (also called distance metrics) for routing protocols: for example, all links have the same cost, the cost of a link depends on its bandwidth capacity, or the cost of a link depends on the bandwidth available.

the client may not be able to obtain the IP address of a mail server and thus cannot communicate with it. In other words, DNS attacks can cause denial of service. As another example, if the DNS mapping for www.cnn.com is compromised, an attacker may be able to direct web browsers looking for the news web site to one that gives out counterfeit news. If the web browser does not authenticate the server, the user may use the counterfeit news as if they were genuine. Some applications (e.g., Unix *rlogin*) use name-based authentication. Attacking DNS could change the name-to-address mapping, and hence may allow an attacker's machine to masquerade as a trusted machine. Our approach for protecting DNS is driven by formal specifications. We develop formal specifications to characterize DNS clients and DNS servers and to define a security goal: A DNS server should only use DNS data that are consistent with those disseminated by the corresponding authoritative sources. We present a DNS wrapper, also characterized by formal specifications, that enforces the security goal.

We call our approach *intrusion tolerance* because it is based on prior work on intrusion detection and fault tolerance. Intrusion detection (e.g., [18, 27, 38, 46]) is a retrofit approach to improve the security of computer systems and networks. Intrusion detection systems detect and report security policy violations. To live with the existing systems and network infrastructures (i.e., the legacy system problem), intrusion detection improves their security with minimal changes to them. Because of the huge costs and the difficulties in building useful yet secure systems, we may not be able to replace the existing (insecure) computer and network systems by secure systems in the near future–or perhaps, never.

In an advanced fault tolerant system, the handling of an error can involve the following steps: error detection, damage assessment, reconfiguration, and recovery. Current attempts at intrusion detection are much less ambitious, typically relying on attack detection that triggers a message to be sent to a security officer. Thereafter it is the responsibility of the human security officer to deal with the situation, e.g., to remove an offending user or site, to request additional audit logs for a particular user, or to save audit logs as evidence. We envision that human intervention at this level will not be feasible for much longer, particularly when long delays for human response have high costs, and attacks may rapidly propagate. Our work is a first step towards an expansive view of intrusion detection, which includes detection of security policy violations, system diagnosis for identifying misbehaving components, and automated response (e.g., system reconfiguration) to

prevent propagation of an attack or to restore the operational status of the system.

The main contributions of this dissertation towards this new goal for intrusion detection are summarized as follows:

- *Presents an intrusion tolerance approach for protecting two key components of network infrastructures, namely routers and domain name systems:* Detection is only a part of the control loop. Our approach includes detection, diagnosis, and response. Formalism is an integral part of our approach, which includes modeling of system components, characterizing system components using formal specifications, and proving properties of the solutions. Most of the existing intrusion detection works are ad-hoc in nature; it is difficult to assess the benefits of deploying those solutions. We hope our work can serve as a stepping stone towards a methodology that employs formalism to achieve a higher level of assurance for detection-based solutions.

- *Presents a first detection-based message authentication scheme*[3]*:* Our message authentication scheme is up to two orders of magnitude faster than an MD5/RSA digital signature scheme. Detection-based approaches are conventionally considered as the second line of defense. We show that when prevention-based approaches are too expensive or restrictive to use, a detection-response approach may be an attractive alternative.

- *Presents techniques and protocols to detect and respond to routers that maliciously drop or misroute packets*[4]*:* This is an initial detection-based approach for protecting routing infrastructures from denial of service attacks. Based on reasonable assumptions, we prove important properties of our protocols regarding soundness (i.e., no false positive), completeness (i.e., no false negative), and responsiveness (i.e., ability to restore the operational status of a network).

- *Presents a wrapper-based solution to protect DNS:* Our security goal for DNS is to ensure that protected DNS servers only use DNS data that are known to be consistent with those disseminated by the corresponding authoritative servers. We employ formal specification to describe DNS servers and our DNS wrapper, used to filter out DNS messages destined for a protected server that may cause violations of our

[3] An earlier version of this message authentication work was published as [14].

[4] An earlier version of this work for protecting routing infrastructures from denial of service was published as [15].

security goal. Based on the specification of the DNS wrapper, we implemented a DNS wrapper prototype and evaluated its performance. Our experimental results show that the DNS wrapper is effective against cache poisoning attacks and certain spoofing attacks, and the wrapper does not have a significant impact on the name server response time and the CPU overhead.

The outline for the rest of this dissertation is as follows. Chapter 2 describes our efficient message authentication scheme for link state routing. Chapter 3 presents our techniques and protocols for protecting routing infrastructures from misbehaving routers that drop packets incorrectly or misroute packets. Chapter 4 describes our scheme for protecting domain name systems. Chapter 5 concludes this dissertation and suggests future work.

Chapter 2

An Efficient Message Authentication Scheme for Link State Routing

2.1 Introduction

Routers exchange routing control packets to share their current states. Based on these control packets, routers construct their routing tables to efficiently forward packets from source to destination. If routing infrastructure components (such as routers or inter-router links) are faulty, misconfigured, or compromised, then the exchange of routing control packets may be affected, resulting in improper or incorrect routing in the network. In particular, the network may suffer from degradation of service, unavailability, or misrouting of packets.

Potential attacks on routing infrastructures can be classified as follows:

- *Packet generation*: A router masquerades as a different router to send erroneous control packets, replays stale control packets, or floods the network with excessive control or data packets.
- *Packet alteration*: A router modifies control or data packets in transit. For example, the cost, the ordering, or the freshness information of control packets may be altered.
- *Packet removal*: A router removes control packets to prevent information about net-

work changes from propagating to other routers, or removes data packets in transit to effect denial of service.

- *Misrouting*: A router misroutes control or data packets so that they will take longer (or forever) to reach their destinations.
- *Breach of confidentiality*: A router eavesdrops data and control packets, or performs traffic analysis.

To protect routing control traffic from some of these threats, approaches that support data authenticity (used to provide both proof of data origin and data integrity), ordering, and freshness of control packets have been proposed. Examples are Perlman's [50, 51] work on link state routing protocols, Finn's [20] report on dynamic routing protocols, Kumar's and Crowcroft's [34] paper on inter-domain routing protocols, Murphy's and Badger's [47] paper on OSPF, Smith's and Garcia-Luna-Aceves's [63] paper on BGP, Hauser's, Przygienda's, and Tsudik's [26] paper on link state routing, Sirois's and Kent's [62] paper on Nimrod, and Smith's, Murthy's, and Garcia-Luna-Aceves's [64] paper on distance vector routing protocols.

This chapter presents an efficient message authentication scheme for protecting control packets in link state routing. Previous work such as [50, 51, 47, 26] either is very expensive computationally or has certain limitations, which will be discussed in Section 2.2. We use a detection-diagnosis-recovery approach, which is intrusion detection (e.g., [18, 38, 46, 27]) augmented with system diagnosis and reconfiguration (e.g., [52]). This approach is also used in Chapter 3 and in Bradley, et al.'s paper [6, 7] on protecting routing infrastructures from routers that incorrectly drop packets and misroute packets. Our main goal is to minimize the cost of performing link state update authentication when the network components function normally, which occurs most of the time. In our scheme, a router r uses a key k and a symmetric-key based data authentication scheme (e.g., a keyed-hash scheme) to sign a link state update. The link state update with the signature is disseminated to all other routers. A receiving router optimistically accepts the routing update as if it were authenticated. At a designated time[1], router r will then release the key k. When the key k arrives, the receiving router verifies the authenticity of the key using a secure and efficient method called hash chaining [36]. Then the verified key will

[1] Section 2.3.4 discusses how to choose a safe key release time so that an attacker cannot use the released key to successfully forge control packets.

be used to verify the authenticity of the link state update using the symmetric-key based data authentication scheme. Note that signature generation and verification can be done using a symmetric-key based data authentication scheme, which is orders of magnitude more efficient than a digital signature scheme. If erroneous routing updates are detected, a distributed diagnosis protocol will be activated to locate the misbehaving routers. Then network reconfiguration will be performed to logically disconnect those routers to restore the operational status of the network.

This chapter is organized as follows: Section 2.2 reviews related work on link state update authentication. Section 2.3 details and analyzes our scheme, called optimistic link state verification. Section 2.4 compares our work with related work, and discusses variations and limitations of our scheme.

2.2 Background: Link State Update Authentication

In link state routing[2], every router constructs link state updates[3] (LSUs) that describe the status of the links incident to the router, and distributes those updates to all other routers. As networks are generally not fully connected, a technique known as *flooding* is used for LSU distribution. When a router receives an LSU that it has not received previously, the router forwards the LSU (essentially) unchanged to its neighbors, except the one from which the LSU was received. To make flooding more robust, a router sends an acknowledgement to the neighbor from which it receives an LSU. If the sender does not receive an acknowledgement after a certain time threshold, it will re-transmit the LSU. Based on the LSU received, a router computes the shortest paths to all destinations. Because those computations are performed independently by all routers on the *same* set of LSUs, networks using link state routing converge to a stable state quickly (as opposed to distance vector routing). To protect routers from using erroneous LSUs to compute their routing tables, data authentication is needed to cope with forged LSU generation and LSU modification. Specifically, an attacker may masquerade as a particular router and generate a forged LSU. Moreover, an LSU may be modified by a compromised intermediate router or an active inter-router link attack.

Data authentication schemes can be broadly classified as symmetric-key based and

[2]Examples of link state routing protocols are OSPF [45], IS-IS [28], a proprietary protocol used in the Internet core system known as SPREAD, and a proprietary routing protocol used in the ARPANET [40].

[3]Link state updates are also called link state advertisements.

asymmetric-key based. In a symmetric-key based data authentication scheme, also called a message authentication code (MAC) scheme, a message is signed and verified using the same key. To use a direct MAC scheme for LSU authentication on a network that has n routers, in the worst case[4], each router needs to maintain $(n-1)$ keys[5] and the network as a whole needs to maintain $O(n^2)$ keys. Moreover, every router would need to sign and to send $(n-1)$ LSUs—one for each router—instead of one LSU as in existing link state routing protocols. Because a router cannot verify the authenticity of LSUs not destined for it, a misbehaving router could send different LSUs to different routers, causing inconsistency problems such as routing loops. Thus a direct MAC scheme for LSU distribution is both insecure as well as expensive in terms of processing and network bandwidth overheads.

Perlman's seminal work [50, 51] uses an asymmetric-key based scheme, also called a digital signature scheme, for data authentication in LSU distribution and public key distribution. In a digital signature scheme, a message is signed using a private key and verified using the corresponding public key. Murphy and Badger [47] proposed a design, based on digital signatures, to securely distribute LSUs and public keys in OSPF. A digital signature scheme seems to be a good candidate for solving the LSU distribution problem—only $O(n)$ key-pairs are needed for the entire network, and a signed LSU can be verified by all routers. However, as pointed out in [47], it may be very expensive[6] to generate and to verify digital signatures. The number of signatures needed to be verified by a router depends on several factors: the number of routers in the network, the grouping of routers into neighborhoods/*areas*[7], the frequencies of link state changes and LSU refreshes, the number of internal and external distinguishing subnets[8], and the particular routing protocol used. In OSPF, because the route to each external subnet is advertised in a separate LSU, there may be tens of thousands of those LSUs. To relieve the performance

[4]Partitioning a large network into neighborhoods could reduce the number of keys stored in a router because (most) routers in different neighborhoods do not need to exchange their link states directly; designated routers of every neighborhood exchange LSUs among themselves.

[5]A direct MAC scheme that does not employ pairwise-keys is not suitable for LSU authentication. Specifically, using a shared key for all routers, or having routers share a key with each of their neighbors and using hop-by-hop LSU authentication cannot protect the network from compromised intermediate routers. The former is used in OSPF version 2[45] cryptographic authentication; routers on a network/subnet uses a secret shared key and a MAC scheme to authenticate routing protocol packets.

[6]Experimental results [47] show that it takes at least 270 microseconds to verify an RSA [57] signature with the 512-bit key size using a SPARC-20 and the GNU MP library.

[7]An area is a set of connected networks, hosts, and routers.

[8]A subnet is internal if the subnet and the router reside in the same autonomous system and external otherwise.

impact, Murphy and Badger suggested a few possibilities: (1) using extra hardware in routers to offload LSU signing and LSU verification; (2) changing the OSPF protocol to reduce the frequency of performing LSU verification by packing external routes from the same area in larger aggregates; (3) verifying LSU signatures periodically or on demand. Our work explored the last option.

Hauser, Przygienda, and Tsudik [26] presented a scheme to reduce the cost of LSU authentication. Their scheme is based on a technique called *hash chains*, which was proposed by Lamport [36]. A hash chain of length ℓ is a list $[H(r), \ldots, H^{\ell-i}(r), \ldots, H^{\ell-1}(r), H^{\ell}(r)]$, where r is a secret quantity, H is a one-way hash function, and $H^{\ell-i+1}(r) = H(H^{\ell-i}(r))$. If $H^{\ell}(r)$ can be sent to a verifier securely, the authenticity of $H^{\ell-i}(r)$ can be verified by applying the function H to $H^{\ell-i}(r)$ i times, where $1 \leq i \leq \ell - 1$. Examples of proposed one-way hash functions are MD5 [56] and SHA [48]. In Hauser, et al.'s scheme[9], two hash chains with different seeds r_{up} and r_{down} are used to represent the *up* and *down* state of a link. The originating router uses its private key to sign a message that includes $H^{\ell}(r_{up})$, $H^{\ell}(r_{down})$, and the current time T and floods the message. A receiving router can verify the authenticity of that message using the public key of the originating router. Let Δ be the time interval between consecutive LSU releases. At time $T + i\Delta$, the originating router releases either $H^{\ell-i}(r_{up})$ or $H^{\ell-i}(r_{down})$, depending on the status of the link. This scheme virtually eliminates the need to perform expensive public-key encryption and decryption. Signing and verifying digital signatures are replaced by applications of a hash function, which are orders of magnitude faster. Despite the cost reduction, there are a few drawbacks to the scheme. First, the scheme cannot efficiently handle multiple-valued link states because the costs of generating, verifying, and storing many hash chains may be higher than those of using digital signatures [26]. The need for multiple-valued link states arises when link costs depend on traffic load, and when a border router advertises link costs for destinations that reside in other *autonomous systems*[10] or in other areas within the same autonomous system. Second, Hauser, et al. showed that the maximum clock skew among routers must be less than 3Δ. Otherwise, an adversary may be able to forge an incorrect LSU that is considered to be fresh and authentic by some routers. Finally, the scheme is not suitable for handling frequent link state changes because the hash chains are

[9]Hauser, et al. also presented a variation of their scheme for a relatively stable network. See [26] for details.

[10]An autonomous system is a group of networks and routers under the control of a single administrative authority.

pre-computed assuming certain fixed time intervals between consecutive LSUs. Choosing Δ to be a very small quantity has two problems—the originating router needs to generate and to store long hash chains, and routers have to be very tightly synchronized (c.f., the clock skew problem discussed above).

2.3 Optimistic Link State Verification

Our approach is inspired by work done in the fault tolerance community—error detection, diagnosis, and recovery, specifically. Our scheme is called optimistic link state verification (OLSV). OLSV, like a digital signature scheme, enables a router, say p, to disseminate a signed LSU that can be verified by other routers. Moreover, no other routers can successfully forge p's LSU. OLSV is much more efficient than a digital signature scheme when the network infrastructure is not compromised (which is the common case), yet OLSV does not have the limitations of other schemes (e.g., [26]), discussed in Section 2.2.

In OLSV, a receiving router optimistically accepts an LSU before it can be verified. At a designated time, the key used to sign this LSU will be released. In Section 2.3.4, we discuss how to choose a safe key release time so that an attacker cannot use the key to successfully forge control packets. When the router receives a key used to authenticate the LSU, it will first verify the authenticity of the key using hash chaining. The verified key is then used to verify the authenticity of the LSU using a MAC scheme. If the verification process detects an erroneous LSU, the receiving router reports from which neighbor(s) that erroneous LSU was received. A distributed diagnosis protocol is activated to identify the misbehaving router(s); we will discuss how link attacks are handled later. Based on the diagnosis result, the misbehaving router(s) are logically removed from the network to restore its operational status.

2.3.1 Assumptions

We consider a network of routers that use a link state routing protocol. We use a graph G to represent the network, with vertices representing routers and edges representing communication links. If two routers share a link, we call them *neighbors*. A router that correctly executes the routing protocol is called a *good router*; otherwise, it is called a *bad router*. A router may be bad due to a software/hardware fault, a misconfiguration, or a malicious attack. A failed/compromised link is called a *bad link*; otherwise, it is called

a *good link*. For example, an attacker may compromise a link between two routers by modifying routing control packets sent over it. An LSU includes several fields: originating router id, sequence number, age, and link state data. The sequence number field is used to provide an ordering among LSUs. With a protected sequence number field, replay and reordering attacks can be detected. The age field is used to support freshness; stale LSUs can be prevented from propagating in the network. A router increments the age field of an LSU before forwarding it to its neighbors. Because the age of an LSU needs to be modified by intermediate routers, the age field is excluded in LSU authentication computation. We make the following assumptions:

1. The network remains connected after the removal of bad routers and bad links.

2. There exists a secure public-key distribution protocol. Perlman [50, 51] and Murphy and Badger [47] proposed security protocols for distributing the public keys of routers. OLSV assumes that every router knows the public keys of all routers.

3. Every router has a local clock, and the maximum clock skew between any two good routers is bounded by a quantity, say ϵ. That is, the difference between the clocks of any two routers is less than or equal to ϵ at any time. A secure network time protocol may be used to synchronize the clocks of routers to bound the maximum clock skew. Moreover, we assume that the ratio of clock rates[11] (or clock frequencies) between the fastest clock and the slowest clock among good routers is bounded by a quantity, say α.

4. The total delay—propagation, queueing, and processing delays—for sending a packet using flooding is bounded by a quantity, say δ.

5. There are no adjacent bad routers. This assumption is used to simplify the description of OLSV. We will discuss how this assumption can be removed in Section 2.4.

6. There exists a one-way hash function. Examples of proposed one-way hash functions are MD5 and SHA. We use H to denote a one-way hash function. Given a random quantity y, it is computationally infeasible to find x such that $y = H(x)$. Moreover,

[11] We use Lamport's definition of clock rate [35]: Let $C(t)$ denote the value of the clock C at physical time t. (A discrete clock can be modeled by a continuous function with an error of up to 1/2 "tick".) If we assume that $C(t)$ is a continuous and differentiable function of t, then the rate of clock C at time t is represented by $dC(t)/dt$.

for a random quantity x, it is computationally infeasible to find an $x' \neq x$ such that $H(x) = H(x')$.

7. There exists a cryptographically strong random number generator: The generated numbers are unpredictable, are uniformly distributed, and have long cycles.

8. A secure digital signature scheme is used. Digital signatures can be generated using a cryptographic hash function and a public-key cipher such as MD5 and RSA. We denote the digital signature of a message m signed using p's private key by $S_p(m)$. Without knowing p's private key, it is computationally infeasible to generate $S_p(m')$ for a new message m'.

9. A secure MAC scheme, which includes a MAC generator $MACG$, is used. Tsudik's [68] keyed-hash scheme and HMAC [33] are examples of MAC schemes. Moreover, they are significantly less expensive than digital signature schemes such as MD5/RSA. We use $MACG_k(m)$ to denote the MAC generated by $MACG$ using a key k on a message m. Without knowing k, it is computationally infeasible to generate $MACG_k(m')$ for a new message m'.

2.3.2 Protocol Overview

Our OLSV protocol is sub-divided into three parts, namely sender, receiver, and recovery. Every router runs a sender process, a receiver process, and a recovery process. The sender process generates keys and uses them to generate a MAC for every LSU. These LSUs and the associated MACs are then flooded to other routers as in existing link state routing protocols. The keys are released to other routers at designated times. Section 2.3.3 details the sender process. The receiver process optimistically accepts LSUs (as if they were authenticated) and uses them to compute the local routing table. When the corresponding keys arrive, the receiver process verifies the authenticity of the LSUs received. Section 2.3.4 details the receiver process. When the receiver process detects an erroneous LSU, the recovery process is activated. A recovery process is responsible for diagnosis and reconfiguration. Diagnosis is used to locate misbehaving routers. Based on the diagnosis results, reconfiguration is used to logically disconnect those misbehaving routers from the network to restore its operational status. Section 2.3.5 details the recovery process. The recovery process is designed to counter router attacks. To counter "active"

link attacks[12], neighboring routers use a MAC scheme to authenticate LSUs forwarded between them[13]. Because a router usually has few neighbors, a secret key can be manually set up or established using a key-exchange protocol for each neighboring router pair, and many existing efficient MAC schemes are applicable to authenticate LSUs sent between neighboring routers. For the sake of clarity, we omit this LSU authentication between neighboring routers in the subsequent description of our protocol.

2.3.3 Sender Process

The sender process generates keys using hash chaining, signs LSUs, and distributes keys and signed LSUs to other routers.

Before a router can sign LSU, the sender process generates a random quantity r and constructs a hash chain of length ℓ using r and a one-way hash function H. Then the sender process composes a *key-chain anchor* (KCA) message that contains the router id, the current time T, and $H^{\ell}(r)$ and signs it with the private key of the router. Then the signed KCA message $(id, T, H^{\ell}(r), S_{id}(id, T, H^{\ell}(r)))$ is distributed to other routers via flooding.

The quantities $H^{\ell-i}(r)$, where $1 \leq i < \ell$, are used as keys to generate MACs for LSUs. A *hash-chained key* (HCK) message $(id, i, H^{\ell-i}(r))$ is released to other routers at time $T + i\Delta$, where Δ is the time interval between consecutive key releases. In fact, the sender process only needs to release an HCK if the corresponding $H^{\ell-i}(r)$ is used to generate a MAC.

To make OLSV secure, $H^{\ell-i}(r)$ is used to generate MACs for LSUs only before time $T + i\Delta - \tau$, where τ is a value that we will derive later. When the sender process wants to send an LSU at time t, where $T + (i-1)\Delta - \tau \leq t < T + i\Delta - \tau$, it uses $H^{\ell-i}(r)$ as the key to generate the MAC. The signed LSU message $(LSU, i, MACG_{H^{\ell-i}(r)}(LSU, i))$ is then flooded to other routers. Figure 2.1 depicts the chronological order of the actions performed by the sender process.

[12] In active link attacks, an attacker may remove, modify, or forge control packets sent over a link.

[13] As we will see, our scheme will still work even if we do not perform additional LSU authentication between neighboring routers. Specifically, a link failure may be viewed as a router failure in OLSV. The routers incident to a failed link will detect the failure and cease the neighbor relationship. Consequently, the failed link will not be used. However, using a MAC scheme to authenticate LSUs sent between neighboring routers can prevent link attacks without affecting the connectivity of the network.

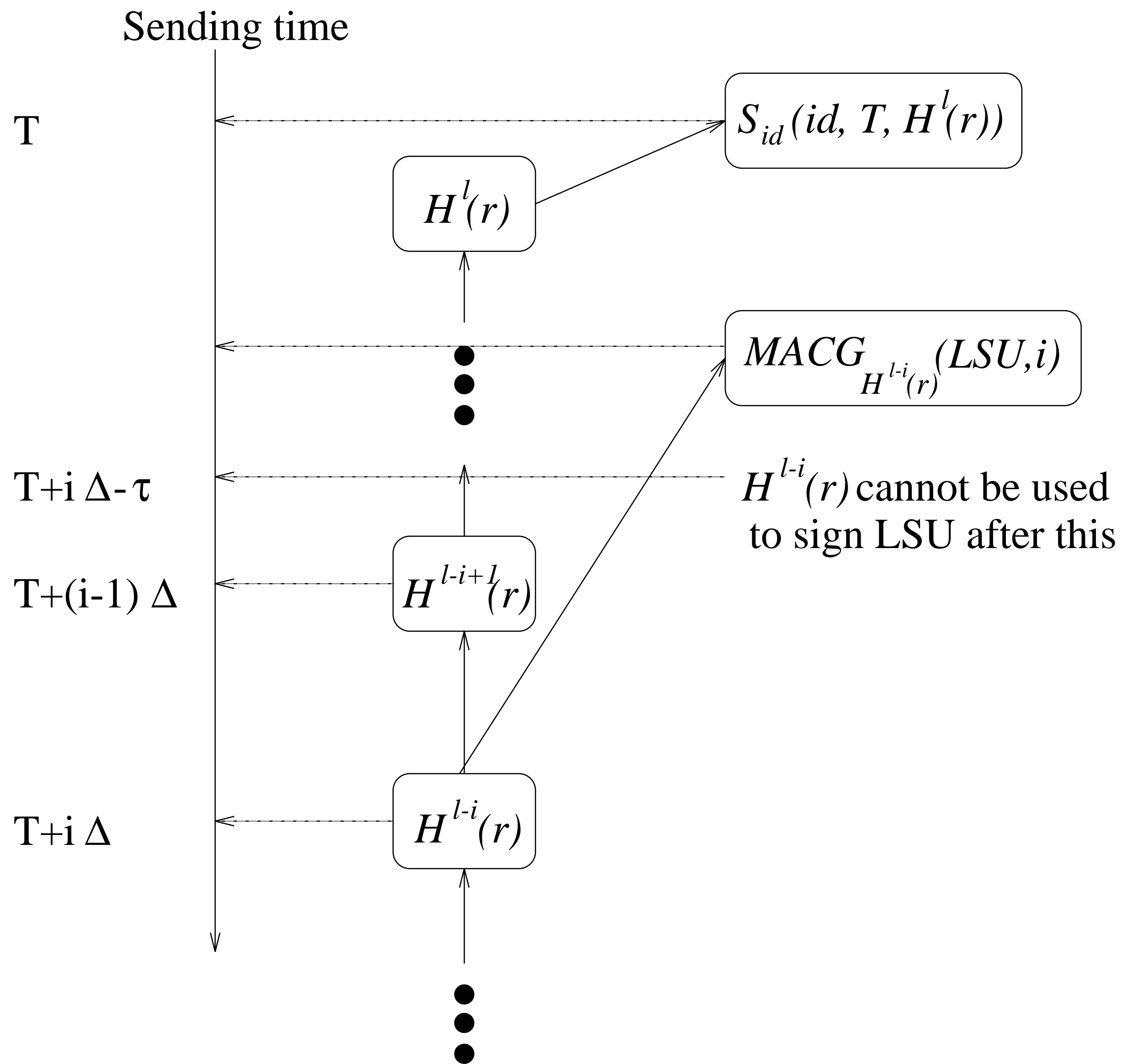

Figure 2.1: The Sender Process.

2.3.4 Receiver Process

The receiver process determines if a signed LSU is acceptable, verifies the authenticity of keys received, and verifies the authenticity of signed LSUs received.

When the receiver process gets a KCA with a digital signature $S_{id}(id, T, H^{\ell}(r))$, it verifies the authenticity of the KCA using the public key of router id. A verified KCA with T "reasonably" close to the current clock value of the router is accepted and stored.

The receiver process optimistically accepts (LSU, i, $MACG_{H^{\ell-i}(r)}(LSU, i)$), a signed LSU, if the receiving time is less than $T + i\Delta - \epsilon$. (Note that the router id in LSU can be used to determine the corresponding T.)

When an HCK message (id, i, k) is received, the authenticity of the HCK is verified by applying the hash function H to k one or more times. Similar to Hauser, et al.'s scheme, the verification process is more efficient if the last verified HCK is stored and used. For example, if the last verified HCK message $(id, i-1, H^{\ell-i+1}(r))$ is stored, then verifying the HCK message (id, i, k) only consists of computing $H(k)$ and comparing $H(k)$ with $H^{\ell-i+1}(r)$. Otherwise, it takes i applications of H to verify the HCK if the KCA (or $H^{\ell}(r)$) is used. A verified HCK message $(id, i, H^{\ell-i}(r))$ is then used to verify the authenticity of LSU. For a signed LSU message (LSU, i, mac), $H^{\ell-i}(r)$ is used to generate the MAC of (LSU, i) and the resulting value is compared to mac. If erroneous LSUs are detected, the recovery process is activated.

Theorem 1 *If we set $\tau \geq 2\epsilon + \alpha\delta$, then an adversary cannot generate an erroneous LSU whose originating router id corresponds to a good router and have the erroneous LSU accepted by good routers without being detected. Moreover, a good router always accepts a signed LSU generated by another good router.*

Proof: Recall that the originating router id releases the HCK message $(id, i, H^{\ell-i}(r))$ at time $T + i\Delta$. At that time, the clock values of good routers are at least $T + i\Delta - \epsilon$. By requiring good routers not to accept LSUs signed with $H^{\ell-i}(r)$ after $T + i\Delta - \epsilon$, the key $H^{\ell-i}(r)$ will not be useful to an adversary to generate erroneous LSUs by the time the adversary receives the HCK message. Note that r is a random quantity and $H^{\ell-j}(r)$, where $i < j < \ell$, are released after the time $H^{\ell-i}(r)$ is released. Moreover, knowing $H^{\ell-k}(r)$, where $1 \leq k < i$, is not useful to determine $H^{\ell-i}(r)$. Thus the adversary cannot determine $H^{\ell-i}(r)$ in time to generate an erroneous LSU and have it accepted by a good

router without being detected. When router id's local time is $T+i\Delta-2\epsilon$, the clock values of good routers are at most $T+i\Delta-\epsilon$. Thus having router id to send the LSU signed with $H^{\ell-i}(r)$ before $T+i\Delta-(2\epsilon+\alpha\delta)$, all good routers will accept the LSU. □

2.3.5 Recovery Process

The recovery process locates misbehaving routers and reconfigures the network to disconnect misbehaving routers from the network.

When erroneous LSUs are detected by the receiver process, a *bad routing update advertisement* (BRUA) message $(BLSU, i, pred)$ is constructed, where $BLSU$ is a "selected" erroneous LSU detected using the key $H^{\ell-i}(r)$, and $pred$ is the id of the neighboring router from which $BLSU$ is received. If multiple erroneous LSUs are detected using $H^{\ell-i}(r)$, the one with the smallest sequence number is selected. Moreover, if more than one erroneous LSUs are associated with the smallest sequence number, the one with the largest checksum is selected. The BRUA is signed with the router's private key and the signed BRUA is then flooded to other routers.

Then the recovery process waits for $2\alpha\delta$ time units to ensure the BRUAs from other routers can reach itself. The $2\alpha\delta$ time delay covers the time for the corresponding HCK and the time for the BRUAs to reach every router. We assume that it takes the same amount of time for every router to use the HCK to verify a LSU and to construct and sign a BRUA. Otherwise, the waiting time should be increased accordingly to include the LSU verification time and the BRUA signing time.

Based on the BRUAs received, the recovery process constructs a *bad routing update propagation* (BRUP) graph. Each BRUP corresponds to one erroneous LSU. In the case where multiple BRUAs corresponding to different erroneous LSUs are received, we use the same arbitration rules—choose the erroneous LSU with the smallest sequence number and use the checksums to break ties—to select one. A BRUA $(BLSU, i, p)$ sent by q is represented by a labeled edge $q \xrightarrow{a} p$ in the BRUP graph, where a is the age of BLSU when q receives it from p.

Then the recovery process performs a depth-first search on the BRUP graph. The search starts with the node that has the largest id and has an outgoing edge. Moreover, if the node has incoming edges, the age value associated with the outgoing edge is larger than those of its incoming edges. Because each node has at most one outgoing edge and the

age values are used to cope with loops, the procedure for finding the starting node is well defined. The search continues until one of the following is encountered: (1) An edge $q \stackrel{a}{\to} p$ and p does not have an outgoing edge; (2) A path segment $q \stackrel{a2}{\to} p \stackrel{a1}{\to} o$, where $a2 \leq a1$; (3) A node visited previously is reached. For the first two cases, (p, q) is recorded. The search procedure is repeated starting with an unvisited node in the BRUP graph.

In case (1), q claims that p sent q the erroneous LSU. Moreover, p does not claim that it received the erroneous LSU from a neighbor. Because link attacks are prevented using a MAC scheme, we can infer that p or q is a bad router. In case (2), q claims that p sent q the erroneous LSU with age a_2. Moreover, p claims that o sent p the erroneous LSU with age a_1. Because p should increment the age field before forwarding the LSU to q, a_2 should be strictly larger than a_1. Thus either p lies or q wrongly accuses p of being a bad router. By a case analysis, one can show that those two cases are sufficient to cover all scenarios in which a bad router sends out the erroneous LSU.

Once bad routers are located, the routers respond by reconfiguring the network to logically remove the bad routers. Specifically, when the diagnosis described above reveals that p or q is a bad router, the neighbor relationship between p and q will be ceased by the good router. (Because we assume that bad routers are not adjacent, either p or q is a good router.) If a bad router keeps sending out erroneous LSUs to its neighbors, the bad router will eventually be disconnected from the network because the connectivity of the bad router is decreased by one every time the router is diagnosed as bad by a neighbor.

2.3.6 An Example

Figure 2.2 depicts a network that has six routers R_i, where $1 \leq i \leq 6$. Consider that R_1 floods a signed LSU $\mathcal{L}$ to other routers. We assume that R_2 is a bad router. Instead of forwarding $\mathcal{L}$, R_2 forwards a modified LSU $\mathcal{L}'$ to its neighbors R_3 and R_5. Without knowing $\mathcal{L}'$ is an erroneous LSU, R_3 in turn forwards $\mathcal{L}'$ to R_6. When the key used to verify $\mathcal{L}$ is disseminated by R_1, the erroneous LSU $\mathcal{L}'$ will be detected. R_6 will send out a BRUA indicating $\mathcal{L}'$ was received from R_3. R_3 and R_5 will send out a BRUA indicating $\mathcal{L}'$ was received from R_2. Then a BRUP graph as shown in Figure 2.2 will be constructed. Performing a DFS on the BRUP graph gives the following results: R_2 or R_3 is a bad router, and R_2 or R_5 is a bad router. Subsequently, R_3 and R_5 will cease their neighbor relationship with R_2. Note that based on the BRUP graph, R_1 cannot determine

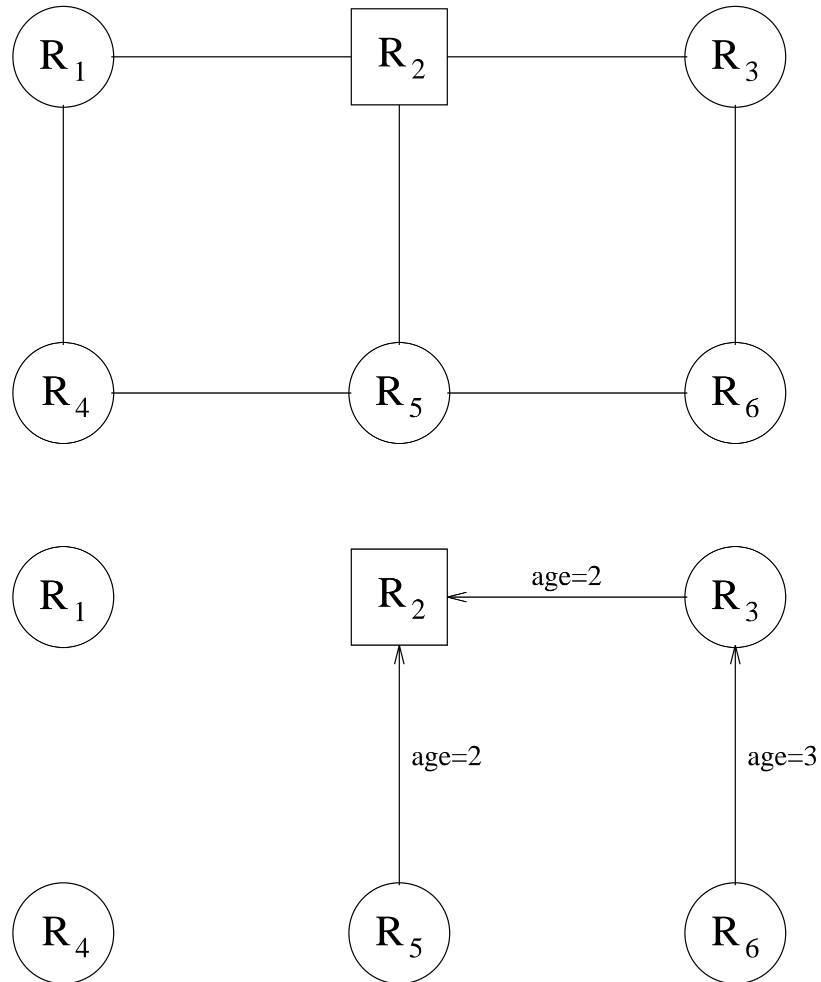

Figure 2.2: A Network and an Associated BRUP Graph.

whether R_2 is a bad router or its neighbors are bad routers. Thus R_1 does not disconnect itself from R_2. However, if R_2 continues to misbehave, R_1 will be able to determine that R_2 is a bad router. As a result, R_1 will cease its neighbor relationship with R_2, and R_2 will be completely disconnected from the network.

2.3.7 Cost Analysis

The major costs of our OLSV scheme are in the time to generate and to verify digital signatures for KCAs and BRUAs, the time to generate and to verify MACs for LSUs, the time to perform DFS on BRUP graphs, the storage required to store hash-chained keys, the storage required to store LSUs that are yet to be verified, and the network bandwidth for sending HCKs, signed LSUs, and BRUAs.

Time: Because a KCA is generated and verified once for every ℓ LSU, the amortized cost is very small. Our experiments, performed on a SPARC-5 running SunOS 4.1.3 and using the RSAREF2 library, show the following results: (1) Generating and verifying a signature for a 16-byte block (i.e., the same size as an MD5 digest) using RSA with the 512-bit key size takes 0.38 second and 0.033 second respectively. (2) Generating an MD5 digest for a 512-byte block and a 16-byte block takes 240 microseconds and 52 microseconds respectively. The time to generate an MD5 digest roughly corresponds to the times to generate and to verify a keyed-MD5 MAC. Thus the overhead of our scheme is up to two orders of magnitude lower than that of an MD5/RSA digital signature scheme for performing LSU authentication. Our main goal is to minimize the performance impact of LSU authentication when the network operates normally, which occurs most of the time. We argue that the cost of recovery (i.e., signing and verifying BRUAs and performing DFS on BRUP graphs), performed only when the network is under attack, is tolerable. Because signing and verifying BRUAs are the dominant factors, the cost of recovery is comparable to that of a digital signature scheme.

Storage: Every router needs to store a hash chain[14] of length ℓ. Note that MD5 produces a 128-bit digest and SHA produces a 160-bit digest. Thus in practice one can choose a large ℓ. On the receiver process side, only the last verified HCK from each router

[14] Note that Hauser, et al.'s scheme requires a router to maintain two hash chains for each link incident on the router. Our scheme only requires one hash chain per router.

needs to be stored. Thus the storage cost for the hash-chained keys may be tolerable. Because a router needs to store the LSUs that are yet to be verified, a potential denial of service vulnerability exists. To prevent a bad router from exhausting the memory of a router by sending out excessive erroneous LSUs, one may impose a limit on how frequently a router may forward LSUs originated by a particular router. In fact, modern link state routing protocols such as OSPF impose a minimum elapsed time between the times a router sends out successive link state advertisements. Another technique is to store conflicting LSUs only—two or more LSUs conflict with each other if they have different link state data but the same originating router id and the same sequence number. When a router receives (erroneous) LSUs whose originating router is the router itself and the sequence number is larger[15] than the router's current sequence number, the router floods an LSU with the same sequence number and the current link state data so other routers can detect a conflict. Note that only conflicting LSUs that have the smallest sequence number are needed to be stored. Moreover, among those LSUs that have the smallest sequence number, only the two LSUs that have the largest checksums are needed. (We need to store two to ensure that the erroneous LSU with the largest checksum is stored; the authentic LSU may have the largest checksum.) As a result, a router only needs to keep an LSU for $2\alpha\delta$ time units (to ensure the LSU from the originating router can reach itself) instead of $\tau + \Delta + \alpha\delta$ time units to detect erroneous LSUs.

Network Bandwidth: The recurring extra network traffic is from HCK messages and two fields in signed LSUs (i.e., an index and a MAC). An HCK and those two fields in a signed LSU are about the size of a message digest. A BRUA is about the size of a signed LSU. Again, a BRUA is sent only when the network is under attack. Thus the extra network bandwidth needed in our scheme should be insignificant.

2.4 Discussion

In this chapter, we present an efficient message authentication scheme, called OLSV, to secure link state routing. OLSV is based on a detection-diagnosis-recovery approach. Previous approaches such as public-key based schemes either are very expensive

[15] Sending erroneous LSUs with old sequence numbers is not an effective attack because those LSUs will not be used.

computationally or have other limitations, which restrict their utility. Our results show that OLSV is up to two orders of magnitude faster than an MD5/RSA digital signature scheme. Consider a network that has 1000 routers. It takes only 0.292 second for a router to verify a LSU (of size 512 bytes) from every router. Thus our scheme is scalable to handle large networks.

Although both Hauser, et al.'s scheme and our OLSV scheme use hash chains as a tool, they differ in how the hash chains are used. In Hauser, et al.'s scheme, hash chain entries are used as signatures. In our OLSV scheme, hash chain entries are used as keys for generating and verifying MACs. Our OLSV scheme has several advantages over Hauser, et al.'s scheme. First, OLSV can efficiently handle multiple-valued link states. The need for multiple-valued link states arises when link costs depend on traffic load, and when a border router advertises (summarized) link costs for destinations that reside in other autonomous systems or in other areas within the same autonomous system. Second, OLSV can be used to handle very frequent link state changes. In OLSV, a hash-chained key can be used to generate and to verify MACs for multiple LSUs. In Hauser, et al.'s scheme, consecutive link state changes are at least Δ time units apart. Moreover, Hauser, et al.'s scheme may not be able to use a small Δ (c.f. Section 2.2). Third, OLSV does not require ϵ to be smaller than a certain value. (Hauser, et al.'s scheme requires $\epsilon < 3\Delta$.) However, we note that there is a tradeoff between the tightness of clock synchronization and the time to recover. It is because a signed LSU may be released $\tau + \Delta$ time units before the time the corresponding hash-chained key is released. A future work item is to reduce the recovery time of OLSV, especially when the routers are very loosely synchronized.

Independently, Wu, et al. [72] proposed an intrusion detection approach to secure link state routing protocols. Their main idea is that a router generates a session key and uses it to sign k LSUs. After the session key is used k times, the originating router will sign the session key using its private key and send the signed session key to other routers. Other routers can verify the authenticity of the session key using the public key of that router. If erroneous LSUs are detected, bad routers are identified using a statistical analysis technique. Even though our OLSV scheme and Wu, et al.'s scheme use a similar approach, our techniques and protocols are quite different. Among other things, our OLSV may be able to detect attacks sooner than Wu, et al.'s scheme. Because generating and verifying digital signatures are expensive, k must be reasonably large. Thus the time between an LSU is sent and the corresponding session key is released may be quite long, especially

when link state changes are infrequent. Our OLSV, on the other hand, does not rely on batch verification for cost reduction. A hash-chained key can be released every Δ time units and Δ may be chosen to be quite small. Thus OLSV may be able to detect erroneous LSUs and initiate the recovery process sooner.

In Section 2.3, we assume that there are no adjacent bad routers in the network. We note that the recovery protocol can be extended to cope with adjacent bad routers. After locating a suspicious router pair by performing DFS in a BRUP graph, we know that at least one of them is misbehaving. If none of them ceases the neighbor relationship, one can conclude that both of them are bad routers. The neighbors of those two routers should disconnect themselves from those two routers in the next round. By repeating this procedure, a good router that is adjacent to those bad routers will be able to determine which neighbors are bad routers and cease its neighbor relationship with them.

Some techniques presented in OLSV are useful in other contexts. The basic idea of optimistic verification is applicable in general to applications in which a subject needs to authenticate data to many other subjects efficiently. For example, it can be used to reduce the costs of Smith's and Garcia-Luna-Aceves's [63] scheme on securing BGP and Smith's, Murthy's and Garcia-Luna-Aceves's [64] scheme on securing distance vector routing protocols. If we remove "optimistic verification" from OLSV, we have an efficient LSU authentication scheme that is applicable to a network with tightly synchronized routers[16]. Specifically, after a router receives a signed LSU, it will wait until the corresponding HCK arrives. The authenticity of the HCK is then verified. Verified HCKs are used to verify the authenticity of LSUs. Only verified LSUs are used to update the routing table. In this case, we would not need to perform the recovery portion of the protocol. Similarly, our techniques are also applicable to some lightweight secure multicast applications outside the domain of routing. For example, contents providers may use our efficient message authentication techniques to securely push news articles and stock quotes to a large number of clients.

[16] Tight synchronization among routers may be achieved using a secure network time protocol.

Chapter 3

Protecting Routing Infrastructures from Denial of Service

3.1 Introduction

To protect a network infrastructure, an efficient and secure message authentication scheme is an important tool to enable secure routing control message exchange. However, a compromised router can cause damage without using incorrect control messages—For example, a misbehaving router may eavesdrop or remove data packets. Tools like Secure Sockets Layer (SSL) are used to protect the privacy and the integrity of data packets. This chapter concerns detecting and responding to some misbehaving routers that cause denial of service.

When a network suffers from denial of service, packets cannot reach their destinations. Existing routing protocols are not well-equipped to cope with denial of service; a misbehaving router—which may be caused by software/hardware faults, misconfiguration, or malicious attacks—may be able to disable entire networks. Because disabling networks can have a huge impact (e.g., time-critical information cannot be communicated) on a large number of people, networks are inviting targets for attack.

In this chapter, we present techniques and protocols for protecting network infrastructures from routers that incorrectly drop packets and misroute packets. Moreover, we prove that our protocols have the following properties:

- A good router never incorrectly claims another router as a misbehaving router.

- If a network has misbehaving routers, one or more of them can be located.
- Misbehaving routers will eventually be removed.

We use a detection-response (i.e., an "expansive" view of intrusion detection) approach for protecting networks from denial of service. In our approach, routers cooperatively diagnose each other to detect, locate, and respond to misbehaving routers. The idea of system diagnosis is not new; Preparata's, Metze's, and Chien's seminal paper [52] proposed a framework for automated system diagnosis. Our contribution is on designing tests specifically for router diagnosis and proving their detection and response properties. Basically, a "testing" router A sends a packet to a "tested" router B and verifies B's behavior against its expected behavior. The verification problem includes two sub-problems: determining B's expected behavior and determining B's actual behavior.

In a network that uses a dynamic routing protocol, B's behavior depends on the current state of the network. Moreover, A and B may not always share the same view of the state. Thus A may not always know the expected behavior of B. We argue that by concentrating on certain types of routing protocols (e.g., in link state routing, unlike distance vector routing, a router propagates routing updates to its neighbors as soon as it receives them) and careful test assignments (e.g., choosing A to be a direct neighbor of B), A and B can see the same network state most of the time. (We will further justify this point in Section 3.8.) This approximation appears to be necessary because of the impossibility of constructing global states of distributed systems.

We assume that A can determine the expected behavior of B and focuses on the second sub-problem (i.e., determining B's actual behavior). There are two basic ways for choosing "test" packets—normal traffic or packets created specifically for testing B. As we will see later, these two strategies give rise to different diagnosis techniques, both of which we consider. If A generates its packets to test B, a major issue is what packets A should generate to uncover the bad behavior of B, if any. Solutions may not exist in all cases. If we assume the worst-case scenario in which B could distinguish ordinary packets from those test packets, B could misbehave only on ordinary packets to avoid being detected. To further complicate the problem, unless the path traversed by a test packet does not involve routers other than A and B, A may need to collaborate with other routers and depend on their reports to diagnose B. Using multiple routers to test a router gives rise to additional issues. First, if A uses reports from misbehaving routers for its analysis, it

may incorrectly deduce that B is a misbehaving router or that B is a good router. Second, the testing routers need to communicate among themselves without being affected by the presence of misbehaving routers in the network.

We believe router diagnosis is an intractable problem if we assume the worst-case adversary. We find that there are special cases that have practical significance and are indeed tractable. We develop a hierarchy of failure models that characterize the behavior and the "strength" of misbehaving routers. For example, routers that misbehave permanently and those that misbehave intermittently are in different failure classes, with the former being a subclass of the latter. Based on those models, we design distributed diagnosis protocols that detect and logically remove misbehaving routers. Once misbehaving routers are located, the routers respond by reconfiguring the network to restore its operational status. It is essential that misbehaving routers cannot misuse the network reconfiguration capability to disconnect good routers or disable the networks. Our protocols solve the misuse problem by only allowing a router to disconnect itself from its neighbors, yet guarantee that all misbehaving routers will eventually be removed.

The outline of this chapter is as follows: To motivate our work, Section 3.2 presents some denial of service examples for computer networks. Section 3.3 reviews related work on securing routing protocols and routers. Section 3.4 describes our system model and failure models for routers. Section 3.5 presents our overall approach for diagnosing routers and the desirable properties of diagnosis protocols. Sections 3.6 and 3.7 detail our techniques and protocols for misbehaving-router detection and present how automated response can be carried out to logically remove those routers, thus restoring the operational status of the networks. Section 3.8 concludes this chapter and discusses the limitations of our work and future work.

3.2 Examples of Routing Infrastructure Failures

In this section, we describe three denial of service examples related to routers and routing protocols. They are the 1980 ARPANET collapse, "black hole" routers, and routers that misroute packets.

In the 1980 ARPANET collapse [58, 20], the source of the problem was mainly due to a faulty router that generated a sequence of erroneous control packets. The sequence numbers of these control packets were of the form $x < y < z$ such that x is more

recent than y, y more recent than z, and z more recent than x. Having these control packets exist simultaneously in a network causes an infinite cycle of control packet acceptance and retransmission. Because routing control packets received a higher priority than the data packets, the routers in the ARPANET spent most of their time handling these routing updates. Thus the network was unavailable for hours. Finn's comments [20] on the ARPANET incident are as follows:

> "It is clear that many such update sequences can be found. This occurred entirely by accident, from an unlikely set of circumstances. Network designers did not consider it a serious possibility. However, a malicious router could easily create this situation and halt the network. Such an attack would be extremely damaging, difficult to prevent, and difficult to correct once it occurred."

Routers exchange control packets to reflect changes, such as topology changes, in a network. A black hole router (e.g., [20]) sends out routing updates claiming that it is on zero-cost (or low-cost) paths[1] to all destinations and then proceeds to drop the packets that it receives. In shortest-path based routing protocols, the most common kind of routing protocols, routers choose the shortest path to forward a packet to its destination. As a result, routers in the neighborhood of a black hole router will direct (some of) their network traffic to the black hole. Figure 3.1 depicts a black hole router. The black hole problem has occurred in operational networks (mostly accidentally) and can cause a widespread denial of service.

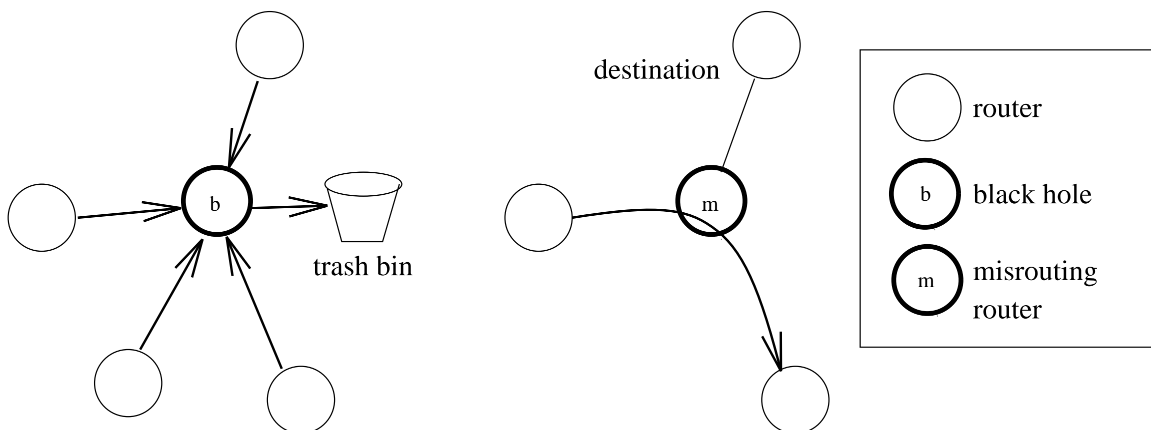

Figure 3.1: Black Hole Routers and Misrouting Routers.

[1]The cost of a path is equal to the sum of the costs of all its constituent links.

Ideally, routers cooperate with each other to deliver the packets to their destinations. However, if the routers make their routing decisions based on different views of the state of the network, routing loops may be formed and the packets caught in them may never reach their destinations. Temporary routing loops occur naturally, say when a link goes down, and solutions have been proposed to deal with them (e.g., [16]). Permanent routing loops or misrouting by a malicious router, depicted in Figure 3.1, are more serious problems. In an IP (Internet Protocol) network, packets have a time-to-live (TTL) field, which guarantees a packet will not stay in the network forever. Hence routing loops and misrouting can cause packets to be dropped and can cause network congestion.

The first example belongs to a family of problems in which routers receive many high priority control packets originating from a router and the routers spend a significant portion of their time processing those packets. This type of problem can be detected and excessive control packets can usually be dropped. For example, some routing protocols (e.g., OSPF) impose a limit on the number of routing control packets originating from a router that are accepted within a specified time interval. No solution has been published to handle black hole routers (c.f., the second example) and misrouting routers (c.f., the third example). We will model these two types of failures and present diagnosis protocols to detect and to respond to them.

3.3 Related Work

Many existing routing protocols are not well protected. For example, sending plain-text passwords in the clear is the only routing update authentication method currently defined in RIP version 2 [39]. Perlman [50, 51] presents a scheme for public-key distribution and for protecting link state updates by means of digital signatures. Finn [20] discusses using public-key and secret-key routing update authentication in general and proposes a secure routing protocol. Kumar and Crowcroft [34] propose a design to secure IDPR, an inter-domain routing protocol. Murphy and Badger [47] propose a design to incorporate public-key distribution and signing link state updates in OSPF. In Chapter 2, we presented an efficient message authentication scheme for link state routing, also based on a detection-response approach. Using strong authentication methods on routing information does not solve all the problems. For example, if a router is faulty or compromised, it may send out erroneous but authentic routing control packets. No prior work has been published

on using a detection-based approach for protecting routing infrastructures from denial of service attacks.

3.4 Our Model

A network is modeled by a directed graph $G = (V, E)$. Vertices represent routers and edges represent communication channels, which may be point-to-point links or networks attached to more than one routers. Note that we do not model hosts that are not routers. If a source host cannot send a packet directly to the destination host, the source will send the packet to a router. We call this router the *source router*. When a router receives a packet, it will send the packet directly to the destination host if it can; otherwise, it will forward the packet to another router "closer" to the destination host. We call the router that delivers the packet to the destination host the *destination router*. A packet generated by a host is represented by a packet generated by the source router. That packet is called a *source packet* with respect to the source router. Moreover, a packet destined for a host is represented by a packet destined for the destination router. That packet is called a *destination packet* with respect to the destination router. Packets processed by a router that are neither source packets nor destination packets with respect to this router are called *transit packets* with respect to this router.

We make the following network assumptions. Assumptions 1, 2, and 3 are used to ensure that a router knows the expected behavior of other routers. Note that Assumptions 1 and 2 can be realized by using link-state routing[2]. We will justify Assumption 1 in Section 3.8.

Assumption 1 (Complete Topology) *Every router has the same map that shows how routers are connected and the cost of communication between neighboring routers.*

Assumption 2 (Shortest-path Routing) *A router always chooses the shortest path to route a packet to its destination.*

Assumption 3 (Bidirectional Channels) $\forall i, j \in V,\ (i, j) \in E \Rightarrow (j, i) \in E$. *In other words, communication channels between neighboring routers are bidirectional.*

[2]As described in Section 2.2, in link-state routing, a router generates an update packet that contains its own identity and the costs (e.g., link delay) to each of its neighboring routers. The update packet is then distributed to all other routers by flooding. Each router collects the update packets from all other routers, constructs the shortest path tree with itself as the root, and updates its own routing table.

Traditionally, security research works on the worst-case assumption that an adversary has unlimited power. Solutions developed under that assumption, if they exist at all, may be impractical to use [41]. In reality, some failures may be less likely to occur than the others. For example, many router failures are caused by accident. Another example is that an attacker may be able to change the routing table of a router but not the router software. Modifying the router software may require detailed knowledge about the routing protocol and the router's operating system, require access to the (possibly proprietary) source code. and require a break-in to a router. In the rest of this section, we present failure models for routers by characterizing the behavior of an adversary. Having those failure models allows us to study the problems and develop solutions for them.

We define a *network sink* as a router that drops (some of) its transit packets. A *black hole* is a network sink that also sends out routing advertisements claiming it can reach certain destinations with costs lower than that it should advertise according to the routing protocol specification. We define a *misrouting router* as a router that forwards a transit packet to a router other than the one on the shortest path to the destination router. A router that exhibits network sink or misrouting behavior is called a *bad router*; otherwise, it is called a *good router*. Bad routers may be caused by software/hardware faults, misconfiguration, or malicious attacks. We make the following assumption about good routers.

Assumption 4 (Existence and Connectivity of Good Routers) *There exists at least one good router in the network and all good routers are connected via good routers.*

We present two independent ways to classify bad routers with the property that a stronger (or more restricted) class is a subset of a weaker (or more general) class. Thus any solution for a weaker class is also applicable to a stronger class. As demonstrated in Sections 3.6 and 3.7, a bad router from a weaker class may be harder to detect than one from a stronger class. Our classifications are depicted in Figure 3.2. The first classification addresses *when* bad routers misbehave, and the second classification addresses on *what packets* bad routers misbehave.

In the first classification, the classes are *permanent*, *almost permanent*, *probabilistic* and *intermittent*. A permanently bad router exhibits the anomalous behavior all the time. An almost permanently bad router is like a permanently bad router, except when it sees explicit control packets associated with a diagnosis procedure: To avoid revealing

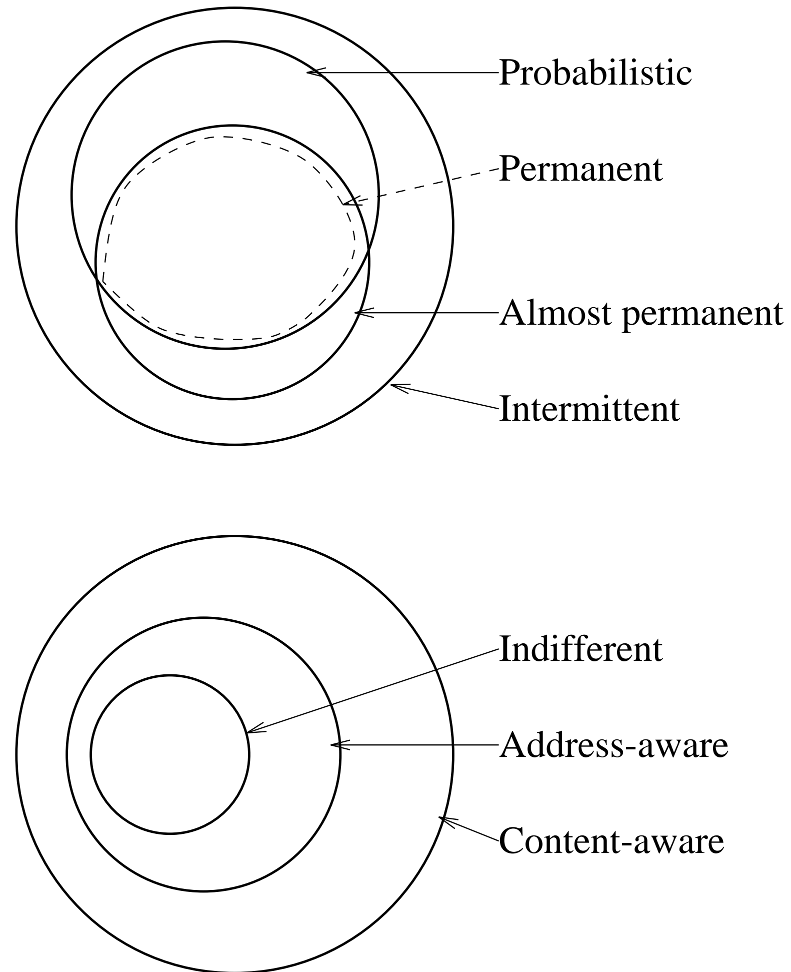

Figure 3.2: Classifications of Bad Routers.

its misbehavior, it may behave like a good router when it is being diagnosed. After the diagnosis is over, it may switch back to the bad-router mode. The almost permanent class represents a way bad routers can "trick" diagnosing routers to cause false negatives. For every transit packet, a probabilistic bad router exhibits the anomalous behavior with a certain probability. An intermittently bad router may exhibit the anomalous behavior at arbitrary times but misbehaves infinitely often. Unlike probabilistic bad routers, conducting statistical tests may not reveal the misbehavior of an intermittently bad router. "Permanent" is weaker than both "almost permanent" and "probabilistic". "Almost permanent" and "probabilistic" are in turn weaker than "intermittent".

In the second classification, the classes, from the strongest to the weakest, use the following criteria to drop or to misroute packets: all packets, the values of source or destination attributes[3] that satisfy certain conditions, and the contents of entire packets, which include the values of the source/destination attributes and the packet payload, that satisfy certain conditions. Those classes are called *indifferent*, *address-aware*, and *content-aware* respectively. In our definition, the "address-aware" class includes the "indifferent" class; an indifferent bad router is a special case of address-aware bad routers in that it does not use the address information. Similarly, an address-aware bad router is a special case of content-aware bad routers. For example, an address-aware bad router may act on packets sent by a certain organization and a content-aware bad router may act on packets that contain certain keywords in their payload.

3.5 Our Approach

In our approach, routers diagnose each other to identify bad routers. Preparata, Metze, and Chien (PMC) [52] proposed a framework for this kind of diagnosis. (Barborak, et al.'s paper [3] surveys extensive work that [52] has initiated.) Preparata, et al. modeled a system equipped with automatic fault diagnosis in which system components can test each other to detect and to locate faulty components. After a component applies a test to another component, the tester will know if the tested component is fault-free or faulty. Permanent faults and perfect test coverage[4] are assumed. In the PMC model, a centralized supervisor is used to collect and analyze all the test results and determine which compo-

[3] For IP networks, the attributes for a source or a destination are an IP address and a port number.

[4] A fault-free tester can always determine accurately the state of a tested component.

nents are faulty. Note that the test results from a faulty component may be unreliable. The PMC model is a starting point for our work, but we need a more realistic model in the context of routing infrastructures.

There are two main issues in our approach. First, given a failure model, we need to design tests that can reveal the anomalous behavior of bad routers. Second, we need to determine how to carry out the diagnosis. In the PMC model, test assignments are designed assuming that a component can test any other component. However, we need to consider the topology of the underlying physical communication network in router diagnosis—how routers can communicate/coordinate with or test each other without being affected by the presence of bad routers. Sections 3.6 and 3.7 present two different techniques for detecting network sinks and misrouting routers, namely distributed probing and flow analysis, and discuss how to perform automated response to reconfigure the network so as to logically remove the bad routers. Distributed probing assumes a more benign bad router model and has a lower cost. Moreover, it works well even if multiple bad routers exist simultaneously. Flow analysis works on a more malicious bad router model; however, it is more expensive because it requires the routers to monitor every transit packet. Our diagnosis protocols are distributed in nature and do not assume a centralized analyzer that gathers and analyses the test results, which may become a single point of failure.

We use the following criteria—the first two concern detection and the third concerns response—to evaluate our diagnosis protocols:

- **Soundness:** If a router is diagnosed as a bad router by good routers, the router is a bad router.
- **Completeness:** If there are bad routers in the network that have misbehaved, our diagnosis routine can locate at least one of them.
- **Responsiveness:** Eventually, all bad routers in the network will be identified and logically removed, and the good routers will still be connected.

3.6 Distributed Probing

In distributed probing, a router diagnoses its neighboring routers by sending them directly (i.e., without passing through intermediate routers) a test packet whose destination router is the tester itself. A tester uses the fact whether it can get back the test packet

within a certain time interval[5] as an indicator of the goodness of the tested router. Note that this test is not useful for all neighboring router pairs. If the shortest path from the tested router to the tester involves other intermediate routers, the fact that the test packet cannot reach the tester does not necessarily mean the tested router is bad.

Distributed probing is applicable to detect network sinks and misrouting routers that cause denial of service—that is, the misrouted packets cannot reach their destinations. In this section, we will present two protocols that detect two different classes of bad routers. The first protocol works for almost permanently, indifferently bad routers. The second protocol works for almost permanently bad routers that are source-address-aware and payload-aware. Before we present our protocols, we will define our notation and state additional assumptions that are specific to distributed probing.

Recall that a network is modeled by a directed graph $G = (V, E)$ where vertices denote routers and edges denote communication channels. Let $e = (i, j) \in E$ be an edge that goes from vertex i to vertex j. The cost[6] of e is denoted by $cost(i, j)$ or $cost(e)$. An edge $(i, j) \in E$ is called *testable* if $cost(j, i)$ is strictly less than the cost of any other path from j to i in G, where the cost of a path is the sum of the costs of all edges on the path. We use Assumption 1 to ensure that i and j see the same network state. The notion of testable edges characterizes the edges useful to distributed probing. Consider a network, depicted in Figure 3.3, that has three routers, namely a, b, and c. We denote $cost(b, c)$, $cost(b, a)$,

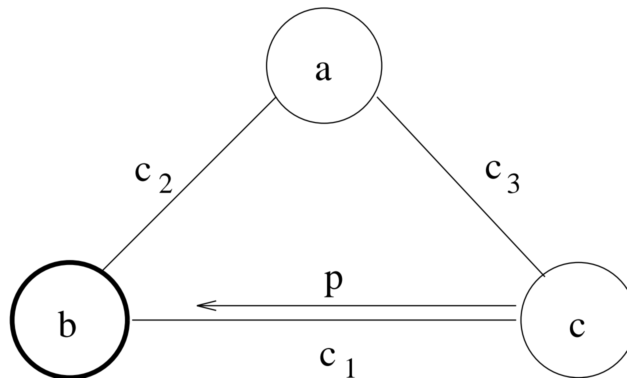

Figure 3.3: Testable Edges.

[5]The time interval is set as an upper bound of the round-trip time between the testing router and the tested neighbor.

[6]The cost (or distance) metric depends on the routing protocol used. For example, every link has the same cost in some routing protocols. On the other hand, the cost of a link depends on the link capacity in others. Our work is applicable to any cost metric.

and $cost(a,c)$ by c_1, c_2, and c_3 respectively. The edge (c,b) is testable if $c_1 < (c_2 + c_3)$. Let p be a packet whose destination is c. If (c,b) is testable and router c sends p to b, then p will return to c if and only if b does not misbehave on p. For $i \in V$, $N(i) = \{j \mid (i,j) \in E\}$ denotes the set of neighbors of vertex i. For $S \subset V$, $N(S) = \{j \notin S \mid (i,j) \in E \ \wedge \ i \in S\}$ denotes the set of neighbors of S, a set of vertices. If the context is clear, we sometimes use i, $i \in V$, to refer to the router represented by vertex i.

Assumption 5 (Positive-Cost Edges) $\forall e \in E$, $cost(e) > 0$.

Assumption 6 (Pairwise Private Addresses) *For all $i \in V$ and $j \in N(i)$, i has an address that i can, but j cannot, reach without using any intermediate routers*[7]*. We call this address the pairwise private address of vertex i with respect to vertex j and denote it by $paddr_j(i)$. This requirement ensures that a testing router can generate a packet whose destination is the testing router itself and the packet is a transit packet for the tested router.*

Protocol 1 (Autonomous Distributed Probing)

$\forall i \in V$, *vertex i executes the following at random times*[8]*:*

For each $j \in N(i)$ *such that* (i,j) *is testable*

Send a packet p whose destination is $paddr_j(i)$ to j via (i,j);

If p *does not return to* i

Then i *ceases its neighbor relationship with* j[9] *(i.e., i believes j is bad)*

Else i *does nothing (i.e., i believes j is good)*

Lemma 1 *Given that bad routers are almost permanent and indifferent, Protocol 1 is sound.*

Proof: Because Protocol 1 does not use any control packets, it is applicable to diagnosing almost permanently bad routers. The soundness of the protocol follows from the definition of testable edges. □

Lemma 2 *Given that bad routers are almost permanent and indifferent, Protocol 1 is complete.*

[7] If necessary, we may assign an unused address to a router interface to realize this requirement.

[8] If we execute the diagnosis protocol at deterministic times, a bad router can avoid revealing its misbehavior by behaving like a good router during these diagnosis time periods.

[9] Broadcasting neighbor relationship changes can be done by flooding, the procedure used by link state protocols to distribute routing updates.

Proof: Consider a maximal connected component of bad routers in G. We denote the set of those bad routers by B. If $N(B)$ is empty, then by Assumptions 3 and 4 the bad routers are disconnected from the network. Otherwise, we claim that at least one vertex in $N(B)$, the set of good neighbors of B, has a testable edge to a vertex in B. On the contrary, we assume that none of the vertices in $N(B)$ has a testable edge to a vertex in B. Let $BN = \{(x, y) \mid x \in B \wedge y \in N(B)\}$, the set of edges incident to a vertex in B and a vertex in $N(B)$. Moreover, let $(b, n) \in BN$ be an edge such that $\forall e \in BN, cost(b, n) \leq cost(e)$. Because we assume (n, b) is not testable, there exists a multi-edge path $P = (b \rightarrow ... \rightarrow n)$ such that $cost(P) \leq cost(b, n)$. Thus, by Assumption 5, $\exists e \in BN$ such that $cost(b, n) > cost(e)$, which contradicts the choice of (b, n). □

Lemma 3 *Given that bad routers are almost permanent and indifferent, Protocol 1 is responsive.*

Proof: Lemma 2 proves that at least one of the bad routers, say b, will be located by a good router, say g. Then, by Protocol 1, g will cease its neighborhood relationship with b. Recall our assumption that the good routers are connected in G. The new graph $G' = (V, E')$ where $E' = E - \{(b, g), (g, b)\}$ has all the good routers remain connected. Note that bad routers' disconnecting themselves from their neighbors, no matter good or bad, does not affect the result. Thus running Protocol 1 continuously will eventually remove all the edges between a good router and a bad router, yet maintaining good routers connected. □

Theorem 2 *Given that bad routers are almost permanent and indifferent, Protocol 1 is sound, complete, and responsive.*

Proof: The proof follows from Lemma 1, Lemma 2, and Lemma 3. □

Protocol 1 can be modified to cope with permanently bad routers that are source-address-aware and payload-aware. A fresh and authenticated diagnosis request that contains the values of source attributes and payload can be distributed to all routers by flooding. Then the routers will use those values to construct their test packets. However, the request may alert the bad routers about the upcoming diagnosis. Thus the flooding of that diagnosis request disqualifies the protocol for detecting almost permanently bad routers. In the following, we present another modification to Protocol 1, which we

call source-initiated distributed probing, that almost avoids the problem of alerting bad routers about the diagnosis, unless the router used to initiate the diagnosis protocol is bad.

In source-initiated distributed probing (Protocol 2), a fresh and authenticated start diagnosis request that contains an identifier, *id*, and the values of source attributes and payload is sent to a router, say r. For example, we can choose the source router with respect to the values of source attributes as r. If a host discovers its packets cannot reach their destinations, it can send a request that contains the information about those packets, which can be used to construct a diagnosis packet, to the source router r. Protocol 2 assumes that r is a good router; otherwise, the protocol may fail. Note that in Protocol 2, a router forwards a start diagnosis request to a neighbor only after that neighbor is diagnosed to be a good router. Thus routers will not be alerted about the diagnosis before they are judged to be good routers. To stop the diagnosis, an authenticated quit diagnosis request that contains *id* will be sent to all routers.

Protocol 2 (Source-initiated Distributed Probing)

$\forall i \in V$, *if* i *receives a fresh and authenticated start request, sr, that contains the values for source attributes,* s, *and the payload,* l, *then* i *executes the following at random times:*

If i *receives the authenticated quit request, qr*

Then i *forwards qr to all neighbors that* i *has sent the corresponding sr;*

i *quits the diagnosis;*

For each $j \in N(i)$ *such that* (i,j) *is testable*

Send a packet p *whose source is* s, *destination is* $paddr_j(i)$, *and payload is* l, *to* j *via* (i,j);

If p *does not return to* i

Then i *ceases its neighbor relationship with* j *(i.e.,* i *believes* j *is bad)*

ElseIf i *has not forwarded sr to* j, *do so (i.e.,* i *believes* j *is good)*

Lemma 4 *Given that bad routers are almost permanent, source-address-aware, and payload-aware, and the first router chosen to initiate the diagnosis is good, Protocol 2 is complete.*

Proof: Let r be the router chosen to initiate the diagnosis and KG be the set of known "good" routers. KG is initialized to $\{r\}$. We will prove the completeness of Protocol 2 by induction on the cardinality of KG.

Base case (i.e., $KG = \{r\}$): Let $MIN = \{i \in N(r) \mid \forall j \in N(r), cost(j,r) \geq cost(i,r)\}$. We claim that r has a testable edge to every vertex in MIN; otherwise, by Assumption 5, it violates the definition of MIN. If all vertices in MIN are good, then the new KG equals the old $KG \cup MIN$ (i.e., the cardinality of KG is increased by at least one). Otherwise, a bad router is located.

Induction step: Consider an arbitrary vertex $g_1 \in KG$. Let $x_1 \in N(g_1) - KG$. If (g_1, x_1) is testable, then either the new KG equals the old $KG \cup \{x_1\}$ or x_1 is diagnosed as a bad router. If (g_1, x_1) is not testable, then $\exists g_2 \in KG \wedge x_2 \in N(g_2) - KG$ such that $cost(x_1 \rightarrow ...x_2 \rightarrow g_2... \rightarrow g_1) \leq cost(x_1, g_1)$. In other words, we have $cost(x_2, g_2) < cost(x_1, g_1)$. Again, if (g_2, x_2) is not testable, then we can apply the same argument and eventually we can find a testable link originating from a vertex in KG. As a result, either the cardinality of KG can be increased by at least one, or a bad router can be located. □

Theorem 3 *Given that bad routers are almost permanent, source-address-aware, and payload-aware, and the first router chosen to initiate the diagnosis is good, Protocol 2 is sound, complete, and responsive.*

Proof: The proof follows from Lemma 4 and the fact that the proofs of the soundness and the responsiveness properties are the same as those of Protocol 1. □

3.7 Flow Analysis

Flow analysis monitors the transit packets flowing in and out of a router to detect abnormal behavior. For each router, the neighbors collaborate to diagnose the router. To enable robust communication among routers, we use flooding to exchange control messages in our protocol. The flooding technique was also used in Perlman's secure routing work [50, 51]. To detect network sinks, the neighbors verify "conservation of transit traffic", depicted in Figure 3.4, by comparing the amount of transit traffic with respect to the tested router going in and that going out of the router. To detect misrouting routers, they verify that the transit packets coming out of the tested router are correctly forwarded. Flow analysis is applicable to bad routers that are intermittent and content-aware. (That is, all failure models discussed in Section 3.4.) In this section, we will first define our notation, and then state additional assumptions that are specific to flow analysis. Finally, we will present our diagnosis protocol and prove its properties.

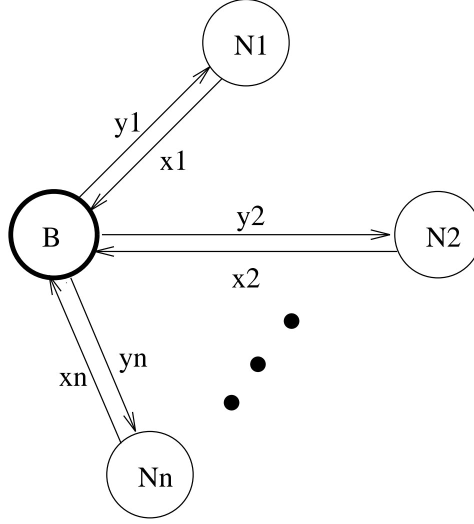

Figure 3.4: Conservation of Transit Traffic: $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$.

For all $(i, j) \in E$ and $k \in \{i, j\}$, let $t_{(i,j)}(k)$ be the accumulated number of bytes of the packet payload[10] for the transit packets with respect to both i and j sent from i to j from k's point of view[11], $n_{(i,j)}(k)$ be the accumulated number of bytes of the packet payload for the packets that are transit to i but non-transit to j sent from i to j from k's point of view, $g_{(i,j)}(k)$ be the accumulated number of bytes of the packet payload for the packets that are source packets of i and transit to j sent from i to j from k's point of view, and $m_{(i,j)}(j)$ be the accumulated number of bytes of the packet payload for the misrouted transit packets with respect to i sent from i to j from j's point of view. Figure 3.5 depicts $t_{(i,j)}(k)$, $n_{(i,j)}(k)$, and $g_{(i,j)}(k)$, which concern packets sent from router i to router j. A router can compute the above counters because of the assumption that routers know the topology of the network and the costs of the edges (i.e., Assumption 1).

Assumption 7 (No Adjacent Bad Routers) *$\forall (i, j) \in E$, i is a good router or j is a good router.*

Assumption 8 (Good Routers Are In The Majority) *The number of good routers $> |V|/2$.*

---

[10]Because of possible packet fragmentation, we use packet payload sizes instead of packet counts. Packet fragmentation occurs because networks have different maximum packet sizes, also known as maximum transfer units (MTU).

[11]We introduce k here to detect disagreements between i and j.

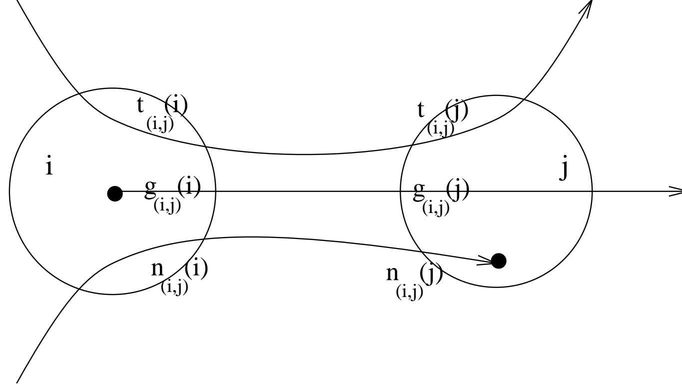

Figure 3.5: $t_{(i,j)}(k)$, $n_{(i,j)}(k)$, and $g_{(i,j)}(k)$, $k \in \{i, j\}$.

Assumption 9 (Negligible Per-hop Packet Delay) *The propagation delay, the processing delay, and the queueing delay for sending a control packet over a link are negligible.*

The execution of Protocol 3 is divided into phases. Every phase has two steps: counter value exchange and distributed diagnosis. In counter value exchange, routers record the values of their counters and broadcast their counter values via flooding to other routers. Protocol 3 would be simpler if we assume the clocks of the routers are synchronized. We do not make that assumption. In Protocol 3, routers synchronize their checkpointing events by exchanging a special message, called the *next phase ready message*. A router broadcasts a next phase ready message when the time elapsed since the last phase change reaches π, the pre-defined time interval between consecutive phases. When a router receives the next phase ready messages from a majority of routers (i.e., $\lceil (|V| + 1)/2 \rceil$ routers), it constructs a *checkpoint message*, which contains the current phase number and its own counter values, and floods the checkpoint message to other routers. Because we assume the majority of the routers are good (Assumption 8), at least one good router is involved in each phase change. Bad routers cannot significantly increase or decrease the time interval between any two consecutive phases. In the protocol, we only require a router and its neighbors to checkpoint at approximately the same time. Because communication channels are bidirectional (Assumption 3), per-hop packet delay is negligible[12] (Assumption 9), and flooding is used, neighboring routers see other routers' next phase ready messages at

[12] We can realize negligible per-hop packet delay by choosing an appropriate π, the pre-defined time interval between consecutive phases.

roughly the same time[13].

Based on the checkpoint messages received, routers perform distributed diagnosis to decide if a neighboring router is bad—not sending out checkpoint messages, sending out erroneous checkpoint messages[14], removing transit packets[15], or misrouting packets. If a router diagnoses a neighboring router as bad, it ceases the neighbor relationship by flooding a link state update that indicates the link between these two routers is down. As a result, routers recompute their routing tables to avoid using that link. To protect the authenticity of the control messages (i.e., next phase ready messages, checkpoint messages, and link state updates), we attach digital signatures to them in Protocol 3 so that the receivers can authenticate the origin and the integrity of these messages.

Protocol 3 (Flow Analysis)

$\forall i \in V$, i initializes the current phase variable, $phase_i$, to zero. A message is called current *if its phase number equals $phase_i$. Let π be the pre-defined time interval between consecutive phases and Δt_i be the local time elapsed since the last phase started. If i has just started, "last phase started" denotes the time i started running the protocol. Then i executes the following:*

> Wait until *(1) $\Delta t_i = \pi$ or (2) $\lceil (|V| + 1)/2 \rceil$ authenticated current next phase ready messages have been received;*
>
> *Broadcast an authenticated next phase ready message that contains $phase_i$;*
>
> Wait until $\lceil (|V| + 1)/2 \rceil$ *authenticated current next phase ready messages have been received;*
>
> *Store and then reset local counters (i.e., $t_{(i,j)}(i), t_{(j,i)}(i), n_{(i,j)}(i), n_{(j,i)}(i), g_{(i,j)}(i)$, $g_{(j,i)}(i)$, and $m_{(j,i)}(i)$);*
>
> *Set $\Delta t_i = 0$;*
>
> *Broadcast an authenticated checkpoint message that contains (1) $phase_i$, (2) $\forall j \in N(i)$, $t_{(i,j)}(i)$, $n_{(i,j)}(i)$, and $g_{(i,j)}(i)$, and (3) $\forall k \in V$ such that $i \in N(k)$, $t_{(k,i)}(i)$, $n_{(k,i)}(i)$, and $g_{(k,i)}(i)$;*
>
> For each $j \in N(i)$

[13] A bad router could delay forwarding packets to its neighbors; however, it can only harm itself (i.e., exposing its misbehavior) for not having a consistent view with its neighbors.

[14] For each neighbor, a router compares its own counter values with the corresponding counter values reported by that neighbor to detect discrepancies.

[15] If a router drops transit packets without sending out erroneous checkpoint messages, the misbehavior will be revealed by the conservation of transit traffic test.

If *j's authenticated current checkpoint message has been received and* $t_{(i,j)}(i) = t_{(i,j)}(j) \wedge n_{(i,j)}(i) = n_{(i,j)}(j) \wedge g_{(i,j)}(i) = g_{(i,j)}(j)$

Then

If $\forall k \in V$ *such that* $k \in N(j)$ *or* $j \in N(k)$,

k's authenticated current checkpoint message has been received and

$(k \in N(j) \Rightarrow (t_{(j,k)}(j) = t_{(j,k)}(k) \wedge n_{(j,k)}(j) = n_{(j,k)}(k) \wedge g_{(j,k)}(j) = g_{(j,k)}(k)))$ *and*

$(j \in N(k) \Rightarrow (t_{(k,j)}(j) = t_{(k,j)}(k) \wedge n_{(k,j)}(j) = n_{(k,j)}(k) \wedge g_{(k,j)}(j) = g_{(k,j)}(k)))$

Then

If $\sum_{l \in \{n \mid j \in N(n)\}} (t_{(l,j)}(j) + g_{(l,j)}(j)) \neq \sum_{l \in N(j)} (t_{(j,l)}(j) + n_{(j,l)}(j))$

(i.e., conservation of transit traffic violated)

Then *i ceases its neighbor relationship with j;*

Else *do nothing (because other routers will respond to the problem);*

Else *i ceases its neighbor relationship with j;*

For each $j \in \{n \mid i \in N(n)\}$

If *j's authenticated current checkpoint message has not been received or* $t_{(j,i)}(i) \neq t_{(j,i)}(j) \vee n_{(j,i)}(i) \neq n_{(j,i)}(j) \vee g_{(j,i)}(i) \neq g_{(j,i)}(j) \vee m_{(j,i)}(i) \neq 0$

Then *i ceases its neighbor relationship with j;*

Set $phase_i = phase_i + 1$

Lemma 5 *Given that bad routers are intermittent and content-aware, Protocol 3 is sound.*

Proof: We prove the lemma by case analysis. First, there are three cases a router, say b, can be diagnosed as a network sink by good routers:

- $\exists i \in V$, a good router, such that $i \in N(b)$ or $b \in N(i)$ and i does not receive authenticated current checkpoint messages from b. Because we assume all good routers are connected, and flooding, which takes negligible time, is used to broadcast checkpoint data, i's not receiving b's checkpoint data implies that b has not sent any.

- $\exists i \in V$, a good router, such that $(i \in N(b) \wedge (t_{(b,i)}(i) \neq t_{(b,i)}(b) \vee n_{(b,i)}(i) \neq n_{(b,i)}(b) \vee g_{(b,i)}(i) \neq g_{(b,i)}(b)))$ or $(b \in N(i) \wedge (t_{(i,b)}(i) \neq t_{(i,b)}(b) \vee n_{(i,b)}(i) \neq n_{(i,b)}(b) \vee g_{(i,b)}(i) \neq g_{(i,b)}(b)))$ implies either b or i has lied. Thus b must be a bad router.

- $\sum_{k \in \{n \mid b \in N(n)\}} (t_{(k,b)}(b) + g_{(k,b)}(b)) \neq \sum_{k \in N(b)} (t_{(b,k)}(b) + n_{(b,k)}(b))$ implies b is a network sink because the amount of transit traffic flowing in b is not equal to that flowing out of b.

Note that b is diagnosed by $i \in N(b)$ as a misrouting router only when $m_{(b,i)}(i) \neq 0$. Hence Protocol 3 is sound. □

Lemma 6 *Given that bad routers are intermittent and content-aware, Protocol 3 is complete.*

Proof: By performing a case analysis similar to that of Lemma 5 together with Assumption 7, one can show that bad routers which have misbehaved and are connected to the network will be located by at least one of their good neighbors in the next phase. □

Lemma 7 *Given that bad routers are intermittent and content-aware, Protocol 3 is responsive.*

Proof: By Assumption 7 and Lemma 5, we know that only edges incident on a good router and a bad router are removed from E. Thus good routers will remain connected. By Lemma 6, when a bad router that is connected to the network misbehaves, it will be located by a good router in the next phase. Together with the fact that an intermittently bad router misbehaves infinitely often, eventually all bad routers will be logically removed from the network. □

Theorem 4 *Given that bad routers are intermittent and content-aware, Protocol 3 is sound, complete, and responsive.*

Proof: The proof follows from Lemma 5, Lemma 6, and Lemma 7. □

3.8 Discussion

This chapter addresses denial of service on routers and routing protocols. We present failure models for routers that characterize the behavior of failed routers, which may be caused by natural faults or malicious attacks. Based on the failure models, we develop techniques and protocols to detect and to respond to misbehaving routers, and prove properties of the protocols, namely soundness, completeness, and responsiveness.

To avoid introducing additional vulnerabilities to the routing infrastructures, we designed our protocols in such a way that a bad router cannot disconnect a good router from the rest of the network and a bad router cannot initiate the diagnosis so often to make all routers spend most of their time executing the protocol (c.f. the flow analysis protocol in Section 3.7). In conclusion, if there is a path between the source and the destination on which all routers are good, our protocols guarantee that the network will eventually be able to deliver packets from the source to the destination.

Our work does not solve the entire denial of service problem of routing infrastructures. This chapter represents a first step to protect routing infrastructures from denial of service using an intrusion detection approach. Issues not addressed include the following:

- *There are router failures not covered by our failure models:* For example, a compromised router may modify the content of a transit packet. Distributed probing can be adapted to handle this problem—a router can check the integrity of test packets after they are sent back by tested routers. Digital signatures may be used to detect packet payload modification; however, generating and verifying digital signatures are too computationally expensive to be applicable. Adapting flow analysis to efficiently diagnose this kind of failure appears to be non-trivial and is a future work item.

- *Link failures are not modeled:* Note that in our protocols, a link failure that results in packet loss may be viewed as a node failure. The routers incident to the link will detect the failure and cease the neighbor relationship. Consequently, the failed link will not be used.

- *Our models do not handle packet loss by good routers:* Good routers may be forced to drop packets because of, for example, network congestion and buffer overflow. A general technique to cope with this problem is to set a threshold on how many packets a good router can drop. To illustrate, we may incorporate the "k-out-of-n" method in our distributed probing protocols: A router is considered good if it can pass k out of n tests. For the flow analysis protocol, we may relax the conservation of transit traffic test by checking whether the difference between the amount of incoming transit traffic and that of the outgoing transit traffic is less than a threshold. A threshold may be learned empirically by identifying a recurring pattern of network traffic. We note that there may be a tradeoff between the detection rate and the false positive

rate: If the threshold is low, there may be many false positives. If the threshold is high, some misbehaving routers (e.g., routers that look for very specific packets to drop) may not be detected. This issue needs further research.

- *Nature of our assumptions:* Our assumptions fall into three categories: network configurations (e.g., network connectivity and routing protocols used), the number and the distribution of bad routers (e.g., good routers are in the majority), and packet propagation (e.g., routers see the same network state). One could extend our results by relaxing the assumptions in the first two categories. For example, Bradley, et al. [6, 7] extended the flow analysis protocol to handle networks that have neighboring bad routers (see below). For the packet propagation category, we made Assumptions 1 and 9 to ensure neighboring routers have the same view. These assumptions enable routers to determine testable edges in the distributed probing protocols, to determine whether a packet is misrouted, and to perform consistent checkpointing in the flow analysis protocol. We argue that Assumption 1 is a reasonable assumption. First, by using flooding to disseminate routing updates (as is done in link-state routing protocols) and checkpoint packets, and requiring communication channels to be bidirectional, neighboring (good) routers see the control packets at almost the same time. Second, as noted in [66], link costs are static, independent of link load, in modern link-state routing protocols. Thus normally link states do not change often. To satisfy Assumption 9, we can use a longer time interval between consecutive phases to reduce the impact of packet propagation delay, which causes slightly different checkpoint times among neighboring routers. To cope with the cases in which Assumptions 1 and 9 do not hold, we can incorporate a threshold on the differences between the counter values of neighboring routers in the flow analysis protocol. Future work is needed to validate the practicality of these assumptions.

- *Our models only consider transit traffic:* In other words, packets sent by source hosts to source routers and those sent by destination routers to destination hosts are not addressed. Our work is useful in containing the damage that can be caused by a router to its source and destination packets. A related issue is that a router may claim that it is directly connected to a local network, thus becoming a source/destination router for that local network. To address this problem, routers can be given a list of potential neighbors and use it to identify those false advertisements.

- *We have not examined the overhead on routers that perform diagnosis:* For distributed probing, a router needs to determine the testable links from itself to its neighboring routers, based on the link state updates received, and then sends (and receives) a test packet to (and from), in the worst case, each of its neighbors. The overhead depends on how often the diagnosis is performed. For flow analysis, there are two main sources of overhead: First, for each transit packet, the router needs to look up a table, which may need to be updated when there is a topology change, and then increments the appropriate counter. Second, at the end of each phase, a router broadcasts an authenticated message to signal that it is ready to advance to the next phase (i.e., next phase ready message), and broadcasts an authenticated message containing the values of its local counters (i.e., checkpoint message). Then routers will verify the goodness of their neighboring routers by computing the amount of transit traffic flowing in and out of those neighbors. We note that message authentication in the flow analysis protocol has the characteristic that a router needs to verify the authenticity of control messages sent by many routers. Thus the OLSV protocol discussed in Chapter 2 may be helpful to reduce the message authentication overhead for flow analysis.

Together with Bradley, et al. [6, 7], we improved the results presented in Section 3.7 and performed a more detailed analysis on the overhead for flow analysis. Specifically, the flow analysis protocol presented in Section 3.7 has a limitation that it cannot cope with adjacent misbehaving routers. Bradley, et al.'s paper solves this problem by introducing destination-specific counters to monitor the traffic flows entering and departing a router. The idea is to perform conservation of flow checks on a per destination router basis. In other words, we check to determine if the amounts of transit traffic for a particular destination going in and going out of a tested router are the same. Consider the following scenario. Suppose router A sends a packet to router D along the $A \rightarrow B \rightarrow C \rightarrow D$ path. Let B and C be misbehaving routers and they collaborate with each other to drop the packet. B can evade detection by the flow analysis protocol presented in Section 3.7 by claiming that C is the destination of the packet (i.e., routers B and C can increase the counters $n_{(B,C)}(B)$ and $n_{(B,C)}(C)$ by the size of the packet). If destination-specific counters are used, router B cannot claim that the packet is for router C anymore. Otherwise, router B cannot pass the conservation flow test for the traffic destined for router D.

Chapter 4

Protecting Domain Name Systems

4.1 Introduction

In this chapter, we present a detection-response approach for protecting domain name systems (DNS). DNS manages a distributed database to support a wide variety of Internet applications such as electronic mail, WWW, and remote login. For example, network applications rely on DNS to translate between host names and IP addresses. A compromise to DNS may cause denial of service (when a client cannot locate the network address of a server) and entity authentication to fail (when host names are used to specify trust relationships among hosts).

In our approach, we define our security goal for DNS—A name server should only use DNS data that are consistent with those disseminated by the name servers that manage the relevant part of the DNS name space. Our approach employs formal specifications to provide assurance for our solution. Formal methods have not been used in connection with an intrusion detection approach. We characterize DNS clients and DNS servers using formal specifications. These specifications reflect the minimal functionalities (or "the greatest lower bound") of these DNS components among existing DNS implementations. Thus if our scheme is strong enough to protect DNS components described by those specifications, it is also strong enough to protect all those DNS implementations.

We characterized a DNS wrapper, which enforces our security goal for DNS, using a formal specification. Our DNS wrapper examines DNS messages entering and departing a protected name server to detect those messages that could lead to violations of our security goal. If the wrapper does not have enough information to determine whether

a DNS message represents an attack, it collaborates with the name servers that manage the relevant part of the DNS name space. If the DNS wrapper cannot verify the data of the DNS message to be trustworthy, the wrapper logs the message and prevents it from reaching the protected name server.

In Section 4.2, we present an example to illustrate how DNS vulnerabilities can affect network security. In Section 4.3, we briefly review the basics of domain name systems. (Readers are referred to [1, 43, 44] for more details about DNS.) In Section 4.4, we describe some known DNS vulnerabilities. In Section 4.5, we survey and evaluate existing approaches for securing DNS, including security enhancements implemented in BIND (in particular BIND release 4.9.5) [71], Cheswick's and Bellovin's *dnsproxy* [13], and DNS security extensions (DNSSEC) [19]. In Section 4.6, we present our detection-diagnosis-response approach for protecting DNS. In Section 4.7, we present our system model. In Section 4.8, we formally specify a DNS wrapper that enforces our security goal for DNS. Based on the DNS wrapper specification, we implemented a DNS wrapper prototype. In Section 4.9, we describe our experiments for evaluating the performance of our implementation, and present our experimental results. Our results show that our DNS wrapper incurs reasonable overheads and is effective against some known DNS attacks.

4.2 A Motivating Example

Human beings use mnemonic names such as cs.ucdavis.edu; however, machines work with numbers such as IP addresses. DNS is used to translate host names to IP addresses. Compromising DNS can change this mapping, and affects applications that use name-based authentication. We use remote login (*rlogin*) as an example to show how DNS vulnerabilities can affect security.

Suppose a user on host A logs into a remote host B. In the first scenario, host A uses DNS to find out the IP address of B. Assume the DNS is compromised and the IP address of another host, say C, is returned. Host C can masquerade as B and collect secret information such as the user's password. In the second scenario, when host B receives a remote login request, it uses DNS to find out the name of the requesting host. An attacker may compromise DNS and have it return the host name of one of the machines trusted by B. In *rlogin*, a user can specify a list of trusted hosts and remote logins from these hosts are automatically granted without checking passwords. In other words, the attacker could

successfully log into host B without knowing the password.

4.3 Overview of DNS

4.3.1 What is DNS?

DNS is a distributed database indexed by *names*. A name is a sequence of characters (e.g., host names or IP addresses). One of the main functions of DNS is to map host names (e.g., cs.ucdavis.edu.) to IP addresses (e.g., 128.120.56.188) and vice versa. The database has a hierarchical structure. Figure 4.1 depicts this tree-structured name space.

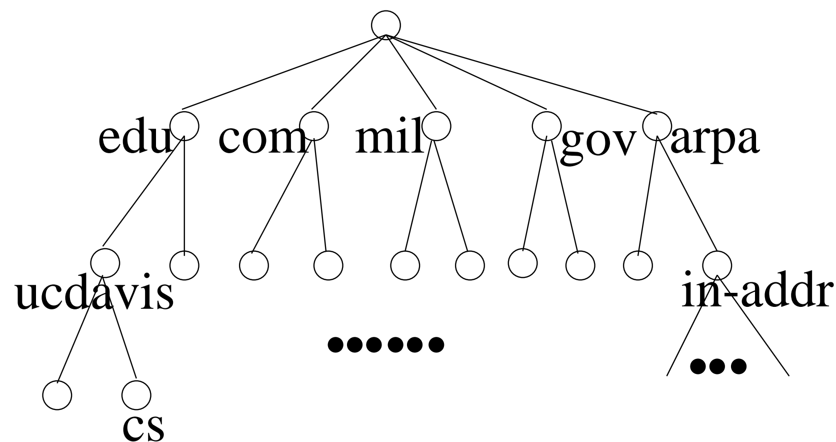

Figure 4.1: Hierarchical Structure of DNS Name Space

A *zone* is a contiguous part of the domain name space that is managed together by a set of machines, called *name servers*. The name of a zone is the concatenation of the node labels on the path from the topmost node of the zone to the root of the domain name space. The set of name servers that manage a zone are said to be *authoritative* for this zone. Every subtree of the domain name space is called a *domain*. The name of a domain is the same as the zone name of the topmost node of the corresponding subtree.

One of the main goals of the design of DNS is to have distributed administration. The distribution is achieved by delegation. For instance, instead of storing all the information about the entire edu domain, which is a very large domain, in a single name server, the responsibility of managing the ucdavis.edu domain is delegated to the authoritative name servers of UC Davis. The authoritative name servers of the edu zone are equipped with the names of the authoritative name servers of the ucdavis.edu zone. Thus if the edu servers

need information about the ucdavis.edu domain, they know which servers to contact.

Clients of DNS are called *resolvers*, which are usually implemented as a set of library routines. Whenever an application on a machine needs to use the name service, it invokes the resolver on its local machine, and the resolver interacts with the name servers to obtain the information needed. The most common implementations of resolvers are called *stub resolvers* (e.g., BIND[1] resolvers are stub resolvers). Stub resolvers only do the minimal job of assembling queries, sending them to servers, and re-sending them if the queries are not answered. Most of the work is carried out by name servers.

4.3.2 How does DNS Work?

The process of retrieving data from DNS is called *name resolution* or simply *resolution*. Suppose the host h1.cs.ucdavis.edu requests the IP address of h2.cs.purdue.edu. The resolver will query a local name server in the cs.ucdavis.edu domain. There are two modes of resolution in DNS: iterative and recursive. In the *iterative mode*, when a name server receives a query for which it does not know the answer, the server will refer the querier to other servers that are more likely to know the answer. Each server is initialized with the addresses of some authoritative servers of the root zone. Moreover, the root servers know the authoritative servers of the second-level domains (e.g., edu domain). Second-level servers know the authoritative servers of third-level domains, and so on. Thus by following the tree structure, the querier can get "closer" to the answer after each referral. Figure 4.2 shows the iterative resolution scenario. For example, when a root server receives an iterative query for the domain name h2.cs.purdue.edu, it refers the querier to the edu servers. In the *recursive mode*, a server either answers the query or finds out the answer by contacting other servers itself and then returns the answer to the querier.

The above resolution process may be quite expensive in terms of resolution time and the number of messages sent. To speed up the process, servers store the results of the previous queries in their *caches*. Consider the above example. If h1.cs.ucdavis.edu asks its local server to resolve the same name twice, the server can reply immediately based on the information stored in its cache the second time. Also, if in a subsequent query h1.cs.ucdavis.edu asks its local server to find out the IP address of h3.cs.purdue.edu, the local server can skip a few steps and contact a cs.purdue.edu server directly. If the querier

[1] BIND stands for Berkeley Internet Name Domain, which is the most common implementation of DNS.

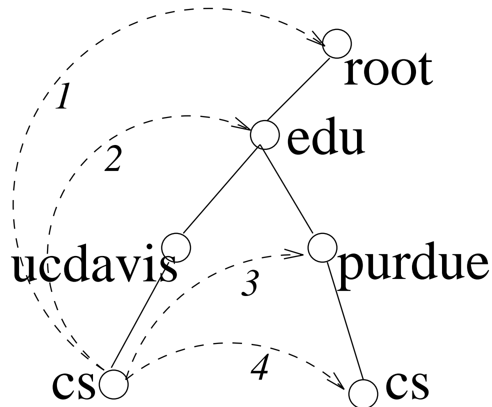

Figure 4.2: Iterative Name Resolution

gets an answer from an authoritative server, the answer is called an *authoritative answer*. Otherwise, it is called a *non-authoritative answer*. Because there may be changes to the mapping, servers do not cache data forever. Authoritative servers attach time-to-live[2] (TTL) tags to data. Upon expiration, a name server should remove the data from its cache.

4.3.3 DNS Message Format

A DNS message consists of a header and four sections: *question*, *answer*, *authority*, and *additional*. A *resource record* (RR) is a unit of information in the last three sections. Here is a list of common resource records [43]:

- An A record contains a 32-bit IP address for the specified domain name.
- A CNAME record lists the canonical name of the specified domain name. In other words, a CNAME resource record maps an alias to the canonical/original domain name.
- An HINFO record identifies the CPU and OS used by a host.

[2]There is no single "best" TTL value for all resource records. The TTL value of a resource record is based on a tradeoff between consistency and performance. A small TTL will increase the average name resolution time because remote name servers will remove the resource record earlier and need to query the corresponding name servers more often. If a resource record is changed, a small TTL enables other name servers to purge the stale data and to use the new data earlier. One should reduce the TTL before the resource record is changed. A common TTL value is one day (e.g., the cs.ucdavis.edu zone), although some high-level zones (e.g., the root zone) use a multi-day TTL.

- An MX record contains a host name acting as a mail exchange for the specified domain.
- An NS record contains a host name that is an authoritative name server for the specified domain.
- A PTR record contains a domain name corresponding to the specified IP address.
- An SOA record indicates the authority for the specified domain name.

The header section includes a query ID[3], an opcode[4], an authoritative answer (AA) flag[5], and a recursion available (RA) flag[6]. The question section carries a target domain name (QNAME), a query type (QTYPE), and a query class (QCLASS). For example, a query to find the IP address of the host h2.cs.purdue.edu has QNAME=h2.cs.purdue.edu, QTYPE=A, and QCLASS=IN (which stands for the Internet). The answer section carries RRs that directly answer the query. The authority section carries RRs that describe other authoritative servers. For instance, the authority section may contain NS RRs to refer the querier to other name servers during iterative resolution. The additional section carries RRs that may be helpful in using the RRs in the other sections. For instance, the additional section of a response may contain A RRs to provide the IP addresses for the NS RRs listed in the authority section.

4.4 DNS Vulnerabilities

Bellovin [4, 5], Gavron [25], Cheswick and Bellovin [12], Schuba and Spafford [60], Vixie [71], and CERT advisory 98:05 [10] have discussed several security problems with DNS. In the following, we summarize their findings: in particular, cache poisoning, failure to authenticate DNS responses, information leakage, masquerading as other name servers, and denial of service.

[3] Query id's are used in both queries and responses to facilitate requesters' matching up responses to outstanding queries.

[4] The opcode field of a DNS message distinguishes between different types of queries—standard queries (SQUERY) and inverse queries (IQUERY). A standard query looks for the resource data given a domain name. An inverse query looks for the domain name given resource data.

[5] The AA flag indicates whether the responding name server is authoritative for the domain name in the question section.

[6] The RA flag indicates whether the responding name server provides recursive resolution services.

In Section 4.2, we described how an old version of *rlogin* could be tricked to believe that the requesting host is a trusted host. Newer versions of *rlogin* are smarter and can cope with this attack. Suppose a host B receives a login request from a machine with IP address IP_a. After B queries DNS to find out the host name corresponding to IP_a, say D, it will perform another DNS lookup to find the IP addresses of D. Then B compares this result with IP_a. This event is logged if IP_a does not match any of these IP addresses of D. This "reverse name lookup" is also known as the Berkeley software patch. To circumvent this cross-checking, an attacker may compromise the name server of the zone in which D is located; however, this may be difficult. In a more sophisticated version of this attack, an attacker poisons the cache of a name server. Recall that a name server caches the results of previous interactions with other servers to improve performance. The attacker can poison the cache of the victim machine by sending DNS packets that contain faked RRs to the victim. By contaminating the cache of B's local server with a mapping entry "D $\rightarrow IP_a$", the attacker can defeat the cross-checking. Schuba's and Spafford's paper [60] described several ways to carry out cache poisoning in great details.

The message authentication mechanism used by most implementations of DNS is weak. Specifically, a querier attaches an id to a query, and uses it to match with the id of the corresponding response. Suppose a server S_1 sends a query to another server S_2. If an attacker can predict the query id used by S_1, a forged response that has a matching query id can be constructed. When S_1 receives the response that claims to be from S_2, S_1 has no way to verify that the response actually comes from S_2. If S_2 is unavailable (e.g., S_2 is under a denial of service attack) when the query is sent, the attacker can just masquerade as S_2 and send the forged response to S_1. Even if S_2 is operational, the attacker may overload S_2 by sending it a lot of queries to cause S_2 to drop S_1's query. Also, if a name server receives multiple responses for its query, it uses the first response. Thus even if S_2 can reply to S_1, the attacker can still succeed if the forged response reaches S_1 before S_2's response does.

Attackers may use DNS to obtain a lot of information about the machines of a domain—for example, host names, machine types, and operating systems used. This information is useful for attackers to choose which machines to attack and which attack methods to use. In BIND release 4.9.3 [70], an access control mechanism is available to restrict which machines can obtain information from a name server.

Gavron [25] described an insecure feature[7] in DNS resolvers regarding their default search heuristic[8]. When an "incomplete" domain name is given to a resolver, it tries to expand it to a "fully qualified domain name". For example, consider a *telnet* attempt from a.b.c.d. to e.f.g.h, which is not a fully qualified domain name because it does not end with a dot. A resolver with this insecure feature will attempt to resolve the domain name of the destination host in the following order: e.f.g.h.b.c.d., e.f.g.h.c.d., e.f.g.h.d., and e.f.g.h.. Thus, if an attacker creates a domain h.d, the traffic would be redirected to an unintended server. In the newer implementations, only the first and the last alternatives—one that falls into the user's domain or one specified by the user—will be tried by a resolver.

CERT [10] published an advisory on several vulnerabilities in BIND that could lead to unauthorized transfers of root privileges and name server crashes. The first vulnerability concerned inadequate checks on the size of inverse queries. (An *inverse query* looks for a domain name given a certain resource/attribute value.) Exploiting this vulnerability could cause malicious data to be written into improper memory locations because of buffer overflow, thus crashing name servers or forcing unauthorized transfers of root privileges. The second vulnerability concerned about inadequate bound checks for processing DNS messages. A malformed DNS message could cause a name server to read from invalid memory locations, which could lead to a system crash. The third vulnerability concerned self-referential CNAME resource records (i.e., resource records that describe a certain domain name as an alias of itself). An attacker could cause denial of service by sending a request that concerned a self-referential CNAME record to a name server.

4.5 Related Work

In this section, we summarize and analyze existing approaches for protecting DNS. Our goal is to present their merits and limitations. Readers are referred to [43, 44] for the specification of DNS. Moreover, Albitz's and Liu's book [1] is an excellent reference for DNS and BIND, the dominant implementation of DNS. Bellovin [5], Schuba and Spafford [60], and Vixie [71] present vulnerabilities of DNS. The two main vulnerabilities studied are cache poisoning and lack of data authentication.

To counter cache poisoning, Vixie [71] presents enhancements to BIND. Briefly,

[7] This feature has been disabled by default in BIND release 4.9.2.

[8] In BIND, the *search* directive can be used to override this default search heuristic.

BIND version 4.9.3 checks the input resource records more carefully before caching them. Moreover, it implements a credibility level scheme in which resource records from a more credible source take precedence over those from a less credible one. Cheswick and Bellovin [13] present a design for a DNS proxy (*dnsproxy*). In their design, the domain name space is partitioned into regions called *realms*. A realm is served by a set of servers. Depending on the query name of a DNS request, *dnsproxy* forwards the request to the servers responsible for the corresponding realm. Certain resource records in response messages—those that do not refer to realm to which the query name belongs, and those that satisfy a set of filtering rules—are removed to protect the queriers. Eastlake and Kaufman [19] present security extensions to DNS (DNSSEC) that uses cryptographic digital signatures to support data authentication for DNS data. In DNSSEC, new resource record types are introduced for public keys and digital signatures. Security-aware servers and security-aware resolvers can use zone keys, which are either statically configured or learned by chaining through zones, to verify the origins of resource records.

Section 4.5.1 describes Vixie's work in hardening BIND. Section 4.5.2 describes *dnsproxy*. Section 4.5.3 describes the DNS security extensions. Section 4.5.4 presents our evaluation for these approaches.

4.5.1 Hardening BIND—Vixie's Approach

Early versions of BIND (pre-BIND 4.9) [71, 5] trust all resource records appearing in responses whose *id* and destination port number match the query *id* and the source port number of an outstanding query. Moreover, the query *id* and the source port number used are quite predictable. Thus these name servers are vulnerable to cache poisoning, and resolvers are vulnerable to getting incorrect information. Vixie [71] presents security enhancements for BIND that fix some of its vulnerabilities.

Vixie's Security Enhancements for BIND

Reference [71] and the source code of BIND 4.9.5 indicate that the following enhancements have been implemented in BIND:

- *Query names stated in responses are matched against those stated in queries sent by name servers and resolvers:* The query name in a response must be identical to that in the corresponding query; otherwise, the response will be dropped.

- *Resource records are filtered before caching:* In old BIND, name servers completely trust and cache all resource records in a DNS response, even those resource records that are not related to the query name. As of BIND 4.9.3, resource records are filtered based on their "relevance" to the query name. A resource record r in the answer section is accepted if r's name is the same as or is a canonical name of the query name. A resource record r in the authority section is rejected if there is a dangling CNAME resource record in the answer section. Moreover, if r is of type NS, r is accepted if (1) the query name or its canonical name belongs to the zone designated by r's name, or (2) there is no CNAME resource record in the answer section and r's name belongs to a domain managed by the responding server. The pseudo code of Vixie's filtering algorithm is shown in Appendix A.

- *Different sources have different credibility levels:* Resource records are tagged with credibility levels depending on how they are obtained (e.g., zone data or authoritative answer). More credible resource records preempt less credible ones from the cache. On the other hand, less credible resource records cannot preempt more credible ones. Credibility levels, from the most credible to the least credible, are zone data, authoritative answers[9], answers[10], additional data[11], and cache data[12].

- *The time-to-live (TTL) fields for resource records in the additional section are decreased more rapidly:* The TTL of additional data—less credible data sent by other name servers—are decreased by 5% after each access. As a result, those resource records will be purged from cache sooner.

- *Abnormal responses are logged:* An abnormal response can be any of the following: (1) Responses coming from an address the server has not queried; (2) Responses containing resource records in the answer section that do not concern the query name or its canonical name; (3) Responses containing bad referral. These log data may be useful for post-mortem analysis.

[9] Resource records in the answer section of a response whose authoritative answer flag is set.

[10] Resource records in the answer section of a response whose authoritative answer flag is clear or those in the authority or additional sections that correspond to a system priming query.

[11] All resource records in a response that are neither authoritative answers nor answers.

[12] Resource records used for root server hints.

Examples of Erroneous Responses Countered by the Enhancements

Suppose a name server queries an authoritative server for hack.com to obtain the IP address of h1.hack.com. (The query is depicted in Figure 4.3.) If the name server of

Header	OPCODE=SQUERY
Question	QNAME=H1.HACK.COM, QCLASS=IN, QTYPE=A
Answer	<empty>
Authority	<empty>
Additional	<empty>

Figure 4.3: Example query.

hack.com is compromised, it may return the response depicted in Figure 4.4. However, the response will be dropped and logged (classified as a malformed response) because of the mismatch between the query name in the question section of the response (i.e., h1.good.edu) and that of the query. In another example, depicted in Figure 4.5, the compromised name

Header	OPCODE=SQUERY, RESPONSE, AA
Question	QNAME=H1.GOOD.EDU, QCLASS=IN, QTYPE=A
Answer	H1.GOOD.EDU 86400 IN A 128.120.56.188
Authority	<empty>
Additional	<empty>

Figure 4.4: Erroneous Response #1.

server attaches erroneous resource records to the response. The resource record in the answer section will be rejected because h2.good.edu is not equal to or a canonical name of h1.hack.com. The resource record in the authority section will also be filtered out because cs.good.edu does not belong to hack.com, the domain managed by the responding server. Let us further assume that our server already has a resource record concerning the IP address of bell.cs.good.edu (i.e., a type A resource record) and that resource record is tagged as an "authoritative answer", the erroneous "A" resource record in the additional section of the response cannot replace the one in cache because of its lower credibility level.

Header	OPCODE=SQUERY, RESPONSE, AA				
Question	QNAME=H1.HACK.COM, QCLASS=IN, QTYPE=A				
Answer	H2.GOOD.EDU	86400	IN	A	128.120.56.189
Authority	CS.GOOD.EDU	86400	IN	NS	BELL.CS.GOOD.EDU
Additional	BELL.CS.GOOD.EDU	86400	IN	A	128.120.55.8

Figure 4.5: Erroneous Response #2.

4.5.2 Dnsproxy

Dnsproxy is an application-level proxy for DNS. Although it is designed to work with firewalls, it can be used standalone. The goal of *dnsproxy* is to control exposure of internal machines to DNS data that come from machines outside the firewall. *Dnsproxy* prevents external (untrusted) name servers from confusing internal hosts about DNS data that concern the "inside" realm—resource records that concern internal domain names, that associate host names to internal IP addresses, or that concern root name servers. As a result, internal machines can depend on name-based authentication for services such as *rlogin, rsh, rcp,* and NFS. Moreover, for sites that manage their own internal DNS, *dnsproxy* can prevent external name servers from confusing internal servers about where the root servers are. In this case, internal servers, except the internal root servers, should never learn the information about the external root servers.

The domain name space is partitioned into the internal realm and the external realm[13]. The former covers the domains that are considered internal or trusted. The external realm covers all other domains. Moreover, a realm is served by a set of name servers. *Dnsproxy* serves as a switch: When *dnsproxy* receives a query, it forwards the query to a server of the corresponding realm. *Dnsproxy* also serves as a filter: When *dnsproxy* receives a response, it checks the response and drops the whole response or some of its resource records if one of the following conditions are met[14]:

- Malformed resource records;
- Mismatches between the query names in the query and the response;

[13] *Dnsproxy* can be extended to handle more than two realms. For example, one can configure *dnsproxy* to have a set of trusted realms and to associate servers to manage these realms. Vixie also suggested a similar idea to lessen the security problem of DNS: "A resolver can be configured with a static map of domains to name server addresses, allowing queries to be forwarded directly to appropriate name servers" (p.211, [71]).

[14] The first three checks are also performed in Vixie's BIND enhancements. (See Section 4.5.1.)

- Mismatches between the expected and the actual IP addresses from which a response comes;
- Resource records concerning the internal realm returned by external servers;
- Resource records that satisfy a resource record blocking rule. The rules used in [13] block all NS records and A records that associate host names to internal IP addresses.

The filtered responses are then returned to the internal queriers. We use an example to illustrate the last two filtering conditions. Suppose the internal realm is defined by the domain good.edu and the corresponding IP network address is 128.120.*.*. When *dnsproxy* receives a query as shown in Figure 4.3, it forwards the query to a server for the external realm because the query name, h1.hack.com, belongs to the external realm. Suppose the external realm server then returns the response as shown in Figure 4.6. The

Header	OPCODE=SQUERY, RESPONSE, AA				
Question	QNAME=H1.HACK.COM, QCLASS=IN, QTYPE=A				
Answer	H1.HACK.COM	86400	IN	A	128.120.56.189
Authority	HACK.COM	86400	IN	NS	NS.HACK.COM
Additional	BELL.CS.GOOD.EDU	86400	IN	A	128.120.55.8

Figure 4.6: Erroneous Response #3.

resource record in the answer section will be dropped because it maps a host name to an internal IP address. The authority record will be removed because all NS records from the external realm are not trusted. Exposing internal name servers to external NS records is dangerous because, depending on the deployment option, internal servers may bypass *dnsproxy* and contact the external servers directly. Finally, the resource record in the additional section will be dropped because it concerns a domain name that belongs to the internal realm.

There are three main configurations to deploy *dnsproxy*. First, we can reconfigure resolvers to send their queries to *dnsproxy* instead of a name server. For BIND, that translates to modifying *resolv.conf* on the affected hosts. A variant of this option is to add a "forwarder" entry in the boot file of *named* to point to the address of *dnsproxy* . If this name server cannot resolve a query, it forwards the query to a host in the forwarder list. Second, we can run *dnsproxy* on internal root server machines. This option assumes that the site maintains its own internal DNS. When a name server cannot resolve a query,

it sends the query to an internal root server. If the query is for the outside realm, the *dnsproxy* running on that root server forwards it to an outside realm server. Third, we can run *dnsproxy* on dynamic packet filters. DNS queries for the external realm are intercepted by dynamic packet filters and then forwarded to *dnsproxy*.

4.5.3 DNS Security Extensions

Security extensions of DNS, called DNSSEC, have been proposed [19, 24] to support DNS data authentication. TIS has implemented a prototype that supports DNSSEC [67]. DNSSEC adds new resource record types[15] to DNS for public keys (KEY) and cryptographic digital signatures (SIG). Signature records are created for DNS data as well as for public keys. A DNS client or a DNS server can use these records to verify the authenticity of the corresponding data.

Each secure zone is associated with a pair of keys[16]—a private zone key and a public zone key. The private key, which must be kept secret, is used to sign the zone's authoritative resource records. The public key is used by other servers to verify the authenticity of signatures generated by this zone. For example, MD5/RSA is a digital signature algorithm used in DNSSEC. A SIG record is created for each record set defined by a domain name-type combination. For example, the zone cs.ucdavis.edu creates a SIG record for all A record(s) of h.cs.ucdavis.edu, which give the IP address(es) of the domain name h.cs.ucdavis.edu. Suppose MD5/RSA is used as the signature algorithm. MD5 is first applied to the resource record set. The hash value is encrypted using the private key of the zone. Then the encrypted value is stored in the SIG record. To support public key distribution, SIG records are also created for public keys of super-zones and sub-zones. For example, the root zone has a KEY record and the corresponding SIG record for the public key of the edu zone. This SIG record is created using the private key of the root zone.

To verify the authenticity of a resource record set, the verifier uses the resource record set, the SIG record, and the verified KEY record for the authoritative zone. Again, suppose the MD5/RSA signature algorithm is used. The KEY record, which contains the

[15]There is another new resource record type proposed in [19], NXT, to allow an authoritative server to declare the non-existence of certain resource records. Because that record type is not related to our discussion, we do not discuss it here. Interested readers may consult [19] for details.

[16]It is possible to have more than one key pairs associated with a zone. For example, when a zone key is changed, there is a transition period during which both the old key pair and the new key pair are used simultaneously.

public key of the authoritative zone, is used to decrypt the value stored in the SIG record. The result is then compared to the result of applying the hash function MD5 to the record set. A match means the record set is authentic.

There are two ways that a server can learn whether a KEY record (i.e., the corresponding public key) is authentic for a zone. First, the server may be configured with the KEY records of a set of trusted zones. If the zone in question is one of these trusted zones, the server knows the authenticity of this KEY record. Second, the server can discover the public keys for other zones through a process called *chaining*. The basic idea is to establish a trusted path from the zone in question to a trusted zone. For example, suppose the server is configured with the public key of the root zone (i.e., the root zone is a trusted zone). Moreover, suppose the root zone has created a SIG record for the KEY record of the edu zone, and the edu zone has created a SIG record for the KEY record of the ucdavis.edu zone. If the server needs to find the authentic KEY record of the ucdavis.edu zone, the server can first find the authentic KEY record of the zone edu and use it to verify the authenticity of the KEY record of the ucdavis.edu zone. This is a recursive process that keeps going until a trusted zone is reached. In this case, the authenticity of the KEY record of the edu zone can be verified using the SIG record generated by the root zone and the KEY record of the root zone.

The specification defines two ways to expire DNS records: time-to-live (TTL) and signature expiration time. The former is the same as that used in the original DNS—When the TTL of a resource record is decremented to zero, this record should be discarded. The latter, which requires that hosts using DNSSEC to have "reasonably" synchronized clocks, specifies an absolute time at which a signature record becomes invalid.

4.5.4 Evaluation

In this section, we evaluate the different aforementioned approaches to protect DNS. Our criteria for evaluation are as follows:

1. *Effectiveness:* Does the proposed solution solve the problems completely? If not, then how much more difficult is it to launch an attack? Is the effectiveness dependent on the security of other sites?

2. *Compatibility:* Does the solution require changes to the DNS protocol? Is it compatible with existing DNS implementations?

3. *Deployment:* How many DNS servers and resolvers need to be changed, especially those in other administrative domains? How much administrative effort is required to implement and tune the proposed scheme (e.g., modifying and installing new software, and reconfiguring existing systems)? Is the solution applicable to specific environments only (e.g., requiring a firewall)?

4. *Scalability:* On the site level, does the solution works for a large site in which there are many name servers? On the Internet level, does it affect the scalability of DNS? For example, suggestions to change DNS from a distributed database system design back to a centralized database system design are unacceptable.

5. *Recurring costs:* How much additional processing, latency, and network bandwidth overheads are incurred?

6. *Solution simplicity:* Is the solution simple to design and to implement?

In the rest of this section, we use the above criteria to evaluate the BIND enhancements [71], *dnsproxy* [13], and the DNS security extensions [19].

Vixie's Hardening BIND Approach

Vixie's solution preserves the existing DNS protocol, is as scalable as the DNS design, incurs minimal recurring costs, and is relatively simple to implement.

The BIND enhancements prevent cache poisoning in some cases as discussed in Section 4.5.1. However, BIND 4.9.5 is still vulnerable to cache poisoning attacks in some other cases. For example, one can trick a name server to accept an arbitrary resource record using an appropriate CNAME[17] resource record in the answer section of a response. BIND 4.9.5 is weak in coping with spoofing attacks. Another problem with the BIND enhancements is that it needs to be deployed Internet-wide to maximize its effectiveness. It is not sufficient to upgrade the BIND server of a site. In the following, we explain these weaknesses of the BIND enhancements in detail.

An attacker can construct an erroneous response message that uses a CNAME resource record in the answer section to introduce arbitrary resource records, as noted in [71]. Specifically, the attacker can use a CNAME record in the answer section of the

[17]Recall that a CNAME resource record contains a domain name which identifies the canonical name of an alias.

response that states the query name is an alias for another domain name. It is important to note that the canonical name can be any domain name. Resource records in the answer section will be accepted if they describe the canonical name. Moreover, resource records in the authority section will be accepted if they are NS[18] resource records of the domains in which the canonical name lives (i.e., the authoritative zone of the canonical name and its ancestor zones). Suppose a server queries another name server for the IP address of *h1.hack.com* (as shown in Figure 4.3). Figure 4.7 depicts an example response that can poison the cache of this server. Because there is a CNAME record that says h2.good.edu

Header	OPCODE=SQUERY, RESPONSE, AA				
Question	QNAME=H1.HACK.COM, QCLASS=IN, QTYPE=A				
Answer	H1.HACK.COM	86400	IN	CNAME	H2.GOOD.EDU
	H2.GOOD.EDU	86400	IN	A	128.120.56.189
Authority	GOOD.EDU	86400	IN	NS	BELL.CS.GOOD.EDU
Additional	BELL.CS.GOOD.EDU	86400	IN	A	128.120.55.8

Figure 4.7: Erroneous Response #4.

is the canonical name of h1.hack.com, the query name, the A record of h2.good.edu in the answer section will be accepted and tagged with credibility level "answer". Moreover, the NS record in the authority section will also be accepted because h2.good.edu belongs to good.edu. [71] claims that the CNAME problem cannot be solved without changing the DNS protocol. The DNS specification [44] specifies that "unsolicited responses or resource records other than that requested" should be discarded. However, the resource records associated with the canonical name are legitimate according to the specification.

Resource records in the additional section are not filtered in BIND 4.9.5 (c.f. Appendix A). There are two defenses employed by BIND to lessen this problem. First, as mentioned in Section 4.5.1, BIND removes certain "additional" records from the cache sooner. Unfortunately, this defense can be circumvented easily because an attack may not require erroneous data to stay in the cache for a long time or an attacker can re-poison the cache after the corresponding entries expire. Second, "additional" records are considered to be less credible than records from more credible sources such as authoritative answers. Thus "additional" records are simply removed when the same data from a more credible source arrive. Furthermore, "additional" records are ignored if the cache has already obtained

[18] Recall that an NS resource record contains a host name corresponding to an authoritative name server for the specified domain.

the data from a more credible source. The effectiveness of this second defense depends on the availability of the data from a more credible source, which cannot be guaranteed. The "segmented cache" proposed in [71] may be used to cope with the "additional" record problem. A segmented cache stores data in different cache segments according to their credibility levels and each cache lookup can specify a set of cache segments to be searched. "Additional" records are stored in a specific segment, which is used to assist resolution (e.g., locating authoritative servers during recursive resolution) but never be used to answer a query. This segmented cache proposal, which involves major changes to the BIND cache design, is not yet implemented.

BIND 4.9.5 is weak in protecting itself from spoofing attacks. To spoof a response, an attacker needs to know the port number and the query *id* used by the querier. BIND servers always use the same port number for receiving TCP or UDP[19] queries and for sending UDP responses. Moreover, BIND servers open a new socket for each TCP request, and BIND clients open a new socket for each TCP or UDP request. As noted in [71], operating systems tend to assign port numbers to sockets using the least-recently-used strategy, which makes port number usage predictable. The query *id* of successive queries used by BIND servers differ by exactly one. Although the query *id* variable can be initialized with a random number at startup time, it is possible to find out the latest query *id* used if an attacker has control over a name server: The attacker can just trick the target name server to query the name server under the attacker's control. Then the attacker can easily predict the next query *id* the target server will use.

Upgrading enhanced local servers alone is not sufficient because they depend on other servers for data over which they do not have authority. Suppose a local server queries an authoritative server of hack.com for the IP address of h1.dept1.hack.com. If that hack.com server is cache-poisoned, it may return an incorrect response concerning its domain and the local server will accept the response. Because of the large number of BIND servers running under different autonomous administrative domains, having all BIND servers on the Internet upgraded may take some time. In addition, many vendors still ship "obsolete, buggy versions of BIND with their operating systems" [59]. There are name servers running on non-Unix platforms (e.g., Windows NT, OS/2, VMS) whose code is derived from BIND or independent of BIND. It probably will take more time for the

[19]BIND queries and responses are usually sent using UDP.

BIND enhancements to be incorporated in them.

The Dnsproxy Approach

Dnsproxy is useful to prevent cache poisoning attacks that concern DNS data related to the internal realm, and is marginally useful for preventing some spoofing attacks when used with a firewall. *Dnsproxy* is compatible with the existing DNS protocol and DNS implementations. Depending on the deployment option, *dnsproxy* ranges from moderate to satisfactory with respect to ease of deployment, and ranges from moderately to satisfactorily scalable. Finally, *dnsproxy* incurs acceptably low recurring costs, and is quite simple to implement.

Dnsproxy protects internal machines from certain cache poisoning attacks. An external server cannot supply DNS data that concern the internal realm (i.e., the protected domain(s)). However, *dnsproxy* may not be useful for DNS data that belong to the external realm; the external realm servers are ordinary DNS name servers that can be cache-poisoned. Moreover, *dnsproxy* may not be useful if an internal server is compromised. It is because erroneous records in responses may not be filtered out, and queries for the internal realm may not pass through *dnsproxy* for performance reasons. When used with firewalls, external machines cannot masquerade as internal realm servers, which live on the other side of the firewalls. Thus *dnsproxy* may prevent spoofing attacks that concern the internal realm.

There are three main options for deploying *dnsproxy*. Those options differ in scalability and ease of deployment.

- *Reconfiguring resolvers to use* dnsproxy *for resolution:* This option is best to protect a small number of machines. In such a case, only these affected machines need to be reconfigured to send their queries to *dnsproxy* instead of a name server.

- *Running* dnsproxy *on internal root servers:* This option is suitable for a large site. If the site has its own internal DNS, we only need to reconfigure the internal root servers to use *dnsproxy*. Not all queries need to pass through *dnsproxy*; internal NS records are passed to internal servers so that they can contact other internal servers directly. All queries concerning the external realm are still processed by *dnsproxy*. This option is more scalable and more efficient than the first one. However, using *dnsproxy* with two DNS trees (i.e., internal and external trees) may cause inconvenience because

machines that live in both sides of the firewall (e.g., portable computers) may have to keep two different configuration files, one for internal root servers and one for external root servers. Moreover, those machines have to determine which set to use for a given query.

- *Running* dnsproxy *on dynamic packet filters:* No reconfiguration of the hosts is necessary. Internal hosts can have the same view as outsiders because NS records from external servers need not be filtered out. Internal servers can contact other internal servers without using *dnsproxy* as an intermediate. Thus, similar to the second option, this option is scalable. However, this option is applicable only if dynamic packet filters that can intercept outgoing DNS queries and pass them to *dnsproxy* are available.

DNSSEC

DNSSEC is moderately effective in coping with cache poisoning attacks, and is very effective for preventing spoofing attacks. DNSSEC requires some changes to the DNS protocol, yet allowing the secure DNS and the original DNS to run concurrently. DNSSEC is moderately easy to deploy, and is as scalable as the original DNS design. However, DNSSEC incurs moderately high recurring costs, and is quite complicated to design and to implement.

Performing data authentication alone is insufficient to solve the cache poisoning problem, which deals with trusts among name servers. A compromised name server can publish authentic yet erroneous DNS data. This may occur if a name server is subverted[20], a private key of the zone is compromised, a zone data file is incorrectly configured, or a system administrator turns bad. Note that it only takes one compromised zone to carry out cache poisoning attacks.

To protect against spoofing attacks, strong cryptographic data authentication is needed. By using signature and zone key records, a DNS server/client can verify the authenticity of resource records, if there is a trusted path from a trusted zone to the zone in question.

DNSSEC, like the original DNS, does not have an explicit revocation mechanism.

[20]The specification [19] recommends the private keys of zones to be kept offline, which tradeoffs security with convenience.

As mentioned in Section 4.5.3, DNSSEC has two ways to expire resource records. A resource record is expired when the time-to-live (TTL) field becomes zero or the expiration time in the signature record is reached. It might be a problem if only the former is used to expire resource records because signature records can only protect the original TTL in those records; the current TTL cannot be protected. As a result, an attacker may be able to replay expired resource records.

DNSSEC incurs moderately high recurring costs. In particular, CPU time is needed to sign and to verify signatures, network bandwidth is needed to transmit additional resource records, and memory is needed to store additional records in name servers.

DNSSEC has similar deployment problems as *dnsproxy* does. The new secure DNS has to be widely deployed to maximize its usefulness. In fact, DNSSEC face more obstacles than *dnsproxy*. First, it may take some time before public key cryptography is widely used. Second, there is additional administrative overhead associated with key management. Third, in addition to protecting server-server communication, resolver-server communication also needs to be protected. There are a lot more DNS clients than name servers. Thus, it probably will take even longer before DNSSEC is widely deployed to protect resolver-server communication.

Summary

We summarize the evaluation of Vixie's BIND enhancements [71], *dnsproxy* [13], and DNSSEC [19] in Table 4.1. We use the following symbols to denote our rating levels: $\otimes$ for excellent; $\oplus$ for good; $\odot$ for neutral; $\ominus$ for bad.

Criteria	**Hardening BIND**	**Dnsproxy**	**DNSSEC**
Effectiveness			
Cache Poisoning	$\oplus$	$\oplus$	$\odot$
Spoofing	$\ominus$	$\ominus/\odot$[21]	$\otimes$
Compatibility	$\otimes$	$\otimes$	$\odot$
Deployment	$\oplus$	$\oplus/\oplus/\odot$[22]	$\odot$
Scalability	$\otimes$	$\odot/\oplus/\oplus$[22]	$\otimes$
Recurring Costs	$\otimes$	$\oplus$	$\odot$
Solution Simplicity	$\otimes$	$\oplus$	$\ominus$

Table 4.1: Summary of Evaluation.

4.6 Our Approach

We present a detection-response approach to protect DNS. The goals for our DNS work are not only to satisfy the aforementioned criteria, but also to provide additional assurance for our solution using formal methods. Our approach consists of the following steps:

- *Declare threats:* Cache poisoning and spoofing.
- *Define our security goal for DNS:* The DNS data used by a name server should be consistent with those disseminated by the corresponding authoritative name servers.
- *Develop DNS model:* We characterize DNS clients and DNS servers using formal specifications.
- *Develop DNS wrapper:* We develop a formal specification for a DNS wrapper that enforces our security goal. Based on the specification, we implement the DNS wrapper.

Our work is inspired by Ko, et al.'s [31, 32] specification-based approach for intrusion detection. In Ko, et al.'s approach, specifications are developed to describe the expected security-related behavior of privileged programs—specifically, behavior related to access control or sequencing of events. Using these specifications as oracles, program executions are monitored to detect exploitations of the vulnerabilities of these programs. Unlike specification-based intrusion detection, our approach is for prevention[23]. In our approach, we characterize a "weakened" behavioral model for a process (e.g., DNS) using formal specifications. To simplify this model, we exclude functionalities of the process that are known to be insecure and those that we wish to strengthen. Moreover, if there are multiple implementations for the process, the model includes only the common functionalities among these implementations. If our solution is able to protect the process characterized by this model, it is strong enough to protect all these implementations. We formally characterize a wrapper that strengthens the process model to achieve our security goal.

[21] It depends on whether firewalls are used with *dnsproxy*.

[22] There are a few options for deploying *dnsproxy* [13]: Reconfigure resolvers to point to a *dnsproxy*; run *dnsproxy* on internal DNS root servers; run *dnsproxy* on dynamic packet filters.

[23] Although our approach is prevention-based, one could use the formal specifications as the basis for an intrusion detection component that protects DNS: When the component detects a suspicious DNS packet, it generates a warning instead of dropping the packet.

The specification for our wrapper addresses trust among name servers, the kind of information that should appear in a DNS response, and message authentication. Our wrapper examines the incoming and the outgoing DNS messages of a protected name server to detect those messages that could cause violations of our security goal. When the wrapper does not have enough information to determine if a DNS message is a threat, it collaborates with the corresponding authoritative name servers to obtain the information needed. When a DNS message is diagnosed to be a possible threat with respect to our security goal, this message is recorded in a security log and dropped instead of being forwarded.

We consider trust among (possibly malicious) network components. Because of scalability reasons, name servers are managed by different administrative domains. A name server should not blindly trust other name servers. Instead, we argue that information from a name server should be trusted only if that server has authority over that information. In some cases, a detection component by itself cannot determine whether a DNS message represents an intrusion; it may have to collaborate with some authoritative sources to perform diagnosis (also called wrapper verification below). This aspect of trust leads to our distributed cooperative detection and response design.

We consider the issue of message authentication, which is important for protecting network services in general. In DNS, when a query is sent to a name server, an attacker could masquerade as that name server and send back a forged response containing incorrect DNS data. The cryptographic extensions proposed in DNSSEC support message authentication for DNS. Our work handles some message authentication attacks, which is useful before DNSSEC is widely deployed.

4.7 System Model

In our model, there are two types of processes: DNS servers and DNS clients (or resolvers). These processes communicate with each other through message passing. Resolvers only communicate with servers; servers can communicate with other servers in addition to communicating with resolvers. There are three kinds of events for a process: message send, message receive, and internal events. These two types of processes are denoted by *Server* and *Resolver* respectively.

$$Server \cup Resolver = Process$$

$$Server \cap Resolver = \emptyset$$

Basically, we model DNS clients and DNS servers as an object that maintains a view on DNS data. The view may be changed only through communicating with other DNS components (i.e., sending DNS requests and receiving DNS responses) or by timeouts for DNS data.

We use the Vienna Development Method (VDM) to specify our system model, because VDM provides a formal language for specifying data and the associated operations, and includes a framework to perform refinements of data and operations. Another reason is that VDM provides a basis for performing formal verification, which makes it more convenient to extend our work in the future. Most of the symbols used in VDM are standard mathematical symbols. We will describe the non-standard or less commonly used ones as we need them. Readers are referred to [29, 2] for more details on VDM.

In the following, Section 4.7.1 presents our DNS data model. Section 4.7.2 formalizes the DNS concepts of trust, authority, and delegation. Sections 4.7.3 and 4.7.4 characterize our DNS client model and our DNS server model respectively. Section 4.7.5 discusses our assumptions about DNS. Section 4.7.6 presents our security goal for DNS.

4.7.1 DNS Data

DNS messages (of type *Msg*) are either a query (of type *Query*) or a response (of type *Resp*).

$$\begin{aligned} Query \cup Resp &= Msg \\ Query \cap Resp &= \emptyset \end{aligned}$$

A message m of type *Msg* consists of the following sections: header, question, answer, authority, and additional. We denote these sections of m by $Hdr(m)$, $Q(m)$, $Ans(m)$, $Auth(m)$, and $Add(m)$ respectively. The header section includes a query ID[24], an opcode[25], a truncated message flag[26], a recursion desired flag[27], and a response code[28]. We denote these fields of m by $id(m)$, $opcode(m)$, $tc(m)$, $rd(m)$, and $rcode(m)$ respectively. The

[24] Recall that query IDs in DNS messages are used to facilitate name servers' matching up responses to outstanding queries.

[25] Recall that the opcode of a DNS message distinguishes between different types of queries—standard queries and inverse queries. A standard query looks for the resource data given a domain name. An inverse query looks for the domain name given resource data.

[26] The truncated message flag indicates whether the DNS message is truncated. Message truncation occurs when the message length is greater than that allowed on the transmission medium.

[27] A querier requests recursive resolution to be used for the query by setting the recursion desired flag.

[28] The response code field is used to indicate errors and exceptions.

question section consists of a domain name, a query type, and a query class. The answer, the authority, and the additional sections consists of resource records (RR). We denote the set of resource records of a message m by $RRof(m)$. A RR consists of a domain name, a type, a class, a 32-bit TTL (in seconds), and a resource data field. For a resource record r, we denote these fields by $dname(r)$, $type(r)$, $class(r)$, $ttl(r)$, and $rdata(r)$ respectively.

DNS manages a distributed database. The database is indexed by a tuple (dname, type, class) of type *Idx*. The range of the database is a set of resource records, abbreviated as RR. To denote this database type in VDM, we use a map type $DbMap : Idx \xrightarrow{m} RR{-}\mathsf{set}$. A *map* type $T = D \xrightarrow{m} R$ has domain D and range R. The domain and the range of T are denoted by $\mathsf{dom}(T)$ and $\mathsf{rng}(T)$ respectively. A map of type T is a set that relates single items in D to single items in R.

$$
\begin{array}{lll}
RRType & = & \{\mathsf{A}, \mathsf{PTR}, \mathsf{NS}, \mathsf{CNAME}, \mathsf{MX}, \mathsf{SOA}, \mathsf{HINFO}, \ldots\} \\
RRClass & = & \{\mathsf{IN}, \ldots\} \\
TTL & = & \{t \in \mathcal{Z} \mid 0 \le t \le 2^{32} - 1\} \\
Idx & :: & \text{dname} : DName \\
 & & \text{type} \ \ : RRType \\
 & & \text{class} \ \ : RRClass \\
RR & :: & \text{dname} : DName \\
 & & \text{type} \ \ : RRType \\
 & & \text{class} \ \ : RRClass \\
 & & \text{ttl} \ \ \ \ : TTL \\
 & & \text{rdata} \ : RData \\
DbMap & = & Idx \xrightarrow{m} RR\text{-}\mathsf{set}
\end{array}
$$

Db represents the data managed by a DNS. *SubDomain* captures the domain-subdomain relationships. Given a domain d, the set of all the sub-domains of d is represented by $SubDomain(d)$. A *zone* contains the domain names and the associated data of a domain, except those that belong to a delegated domain. A zone is a contiguous part of the domain name space that is managed together by a set of name servers. A zone may have a set of delegated subzones, represented by the function *SubZone*. (In VDM, a *function* specification consists of two parts. The first part defines the argument types and the result type, which are separated by the symbol "→". The second part gives the function definition.) For a zone z, $ZoneData(z)$ contains all resource records whose domain names belong to zone z, the *zone cut data*, and the *glue data*. The zone cut data describe the cuts around the bottom of zone z: In particular the NS resource records of the name servers for the delegated zones of z. If there are name servers for the delegated zones residing below

the zone cut, the glue data contain the addresses of these servers.

$$Db : DbMap$$
$$SubDomain : Domain \xrightarrow{m} Domain\text{−set}$$
$$ZoneData : Zone \xrightarrow{m} DbMap$$
$$SubZone : Zone \rightarrow Zone\text{−set}$$
$$\forall z \in Zone \cdot SubZone(z) \triangleq$$
$$\{cz \mid \exists rr \in \mathsf{rng} ZoneData(z) \cdot type(rr) = \mathsf{NS} \wedge dname(rr) \neq z \wedge cz = dname(rr)\}$$

Given a domain name d, a resource record class cr, and a set of resource records rrs, $Canonical(d, cr, rrs)$ returns a set of canonical names for d of class cr that can be deduced from the resource record set rrs. We use a recursive definition for *Canonical* because DNS allows an alias for a domain name to have an alias.

$$Canonical : DName \times RRClass \times RR\text{−set} \rightarrow DName\text{−set}$$
$$\forall\ d \in DName,\ rc \in RRClass,\ rrs \in RR\text{−set} \cdot Canonical(d, rc, rrs) \triangleq$$
$$\{c \mid (c = d) \vee (\exists\ crr \in rrs \cdot dname(crr) = d \wedge type(crr) = \mathsf{CNAME} \wedge$$
$$class(crr) = cr \wedge c = rdata(crr)) \vee$$
$$(\exists\ i \in DName \cdot i \in Canonical(d, cr, rss) \wedge c \in Canonical(i, cr, rss))\}$$

Every process maintains its view of the database. The view of a server, say s, can be partitioned into the authority part (denoted by $View_{auth}(s)$) and the cache part (denoted by $View_{cache}(s)$), where the former takes precedence over the latter. The *map overwrite* operator † takes two map operands and returns a map that contains all the elements in the second operand and those in the first operand whose domain does not appear in the domain of the second operand. For a server that is not authoritative for any part of the database and for a resolver, the corresponding $View_{auth}$ is $\emptyset$.

$$View_{auth} : Process \xrightarrow{m} DbMap$$
$$View_{cache} : Process \xrightarrow{m} DbMap$$
$$View : Process \rightarrow DbMap$$
$$\forall p \in Process \cdot View(p) \triangleq View_{cache}(p) \dagger View_{auth}(p)$$

The function *Fresh* gives the part of a DbMap that corresponds to resource records with positive TTL.

$$Fresh : DbMap \rightarrow DbMap$$
$$\forall dm \in DbMap, i \in dom(Db) \cdot Fresh(dm)(i) \triangleq$$
$$\mathsf{if}\ \forall rr \in dm(i) \Rightarrow ttl(rr) > 0\ \mathsf{then}\ dm(i)\ \mathsf{else}\ \emptyset$$

4.7.2 Trust, Authority, and Delegation

A DNS process may trust a set of servers by accepting all the messages sent by those servers. For example, a stub resolver trusts all DNS data it obtains from a name server. We denote this kind of trust relationship by TS.

$$TS : Process \xrightarrow{m} Server{-}\mathsf{set}$$

Some servers are said to be authoritative for a zone; their views on the zone data define them. $AuthServer$ maps a zone to the list of authoritative servers. $AuthAnswer$ defines the mapping from an index to the *authoritative answer*, defined by the view of the an authoritative server on the index. $Authoritative$ returns true if and only if every resource record in the input resource record set is authoritative.

$$\begin{aligned}
&AuthServer : Zone \xrightarrow{m} Server{-}\mathsf{set}\\
&AuthAnswer : Idx \rightarrow RR{-}\mathsf{set}\\
&\forall i \in dom(Db) \cdot AuthAnswer(i) \ =\\
&\quad \mathsf{let}\ z \in Zone \wedge p \in Process \wedge i \in \mathsf{dom} ZoneData(z) \wedge p \in AuthServer(z)\ \mathsf{in}\\
&\quad View_{auth}(p)(i)\\
&Authoritative : RR{-}\mathsf{set} \rightarrow Boolean\\
&\forall rrs \in RR{-}\mathsf{set} \cdot Authoritative(rrs) =\\
&\quad \forall rr \in rrs \cdot rr \in AuthAnswer((dname(rr), type(rr), class(rr))
\end{aligned}$$

For a server s, $ZoneDelegated(s)$ is defined to be the set of zones delegated by s.

$$\begin{aligned}
&ZoneDelegated : Server \rightarrow Zone{-}\mathsf{set}\\
&ZoneDelegated(s : Server) \equiv \{cz \mid \exists z \in Zone \cdot s \in AuthServer(z) \wedge cz \in SubZone(z)\}
\end{aligned}$$

4.7.3 Resolvers

A resolver is a DNS client that creates queries and sends them to a name server. In our model, a resolver represents both an application and the DNS resolver invoked by the application. Most DNS implementations (e.g., BIND) use stub resolvers, which rely on the servers to resolve a name by means of recursive resolution. Our resolver model is based on BIND resolvers. Resolvers are configured with a static list of DNS servers that provide recursive resolution service. Resolvers trust the RRs of the response messages sent by these servers. This set of servers is used to initialize the trusted server mapping TS. Note

that a BIND resolver, implemented as library routines, does not use query results from previous interactions with name servers to resolve the current query (i.e., BIND resolvers are memory-less). We model this aspect by resetting $View_{cache}$ to $\emptyset$ after a new query is sent by the resolver.

There are three operations for a resolver—initializing the resolver (denoted by $Init_r$), sending a (recursive) query to a server (denoted by $SendQ_r$) and receiving a response from a server (denoted by $ReceiveR_r$). (The subscript r stands for "resolver".) Consider a BIND resolver br. The protocol for a resolver can be described as follows:

Call $Init_r$ with the addresses of a list of trusted servers;

while true

  Call $SendQ_r$ to send a query q to a trusted server for recursive name resolution;

  Call $ReceiveR_r$ to receive a reply for q

An *operation specification* names the operation, specifies the parameters, and specifies the result returned. The next part of the specification, indicated by the keyword ext, defines the state components that will be accessed by this operation. The keywords wr and rd specify whether the operation has read-write or read-only accesses to those state components. The pre-condition and the post-condition of the operation are indicated by the keywords pre and post. The pre-condition of an operation says that the operation is defined if the input satisfies the predicate stated in the pre-condition. If the pre-condition is satisfied before the operation is executed, the results of the operation will satisfy the predicate stated in the post-condition after the operation is executed.

The specifications for the operations for resolvers—initialization, sending queries, and receiving responses are shown below. $Init_r$ initializes a resolver, say br, with a list of servers to be used in name resolution and initializes $View_{auth}(br)$ and $View_{cache}(br)$ to empty sets. $SendQ_r$ sends a query to a trusted name server (in $TS(br)$). The post-condition of $SendQ_r$ models the memory-less aspect of BIND resolvers. $ReceiveR_r$ receives a response m from a name server s. If s is a trusted server and the response m answers

the query that has been asked, $View_{cache}(br)$ will be extended to include the RRs of m.

$Init_r(ss : Server\text{--}\mathsf{set})$
ext wr : $View_{auth}(br),\ View_{cache}(br),\ TS(br)$
pre : true
post : $View_{auth}(br) = \emptyset \wedge View_{cache}(br) = \emptyset \wedge TS(br) = ss$

$SendQ_r(q : Query,\ s : Server)$
ext rd : $TS(br)$
ext wr : $View_{cache}(br)$
pre : $s \in TS(br)$
post : $View_{cache}(br) = \emptyset$

$ReceiveR_r(q : Query,\ m : Resp,\ s : Server)\ r : RR\text{--}\mathsf{set}$
ext rd : $TS(br)$
ext wr : $View_{cache}(br)$
pre : $s \in TS(br) \wedge Q(q) = Q(m)$
post : $View_{cache}(br) = \{(i \rightarrow rrs) \mid \exists rr_1 \in RRof(m)\cdot$
$i = (dname(rr_1), type(rr_1), class(rr_1)) \wedge\ rrs = \{rr_2 \mid rr_2 \in RRof(m) \wedge$
$i = (dname(rr_2), type(rr_2), class(rr_2))\}\}$

4.7.4 Name Servers

Recall that a name server may provide two types of name resolution services, namely iterative resolution and recursive resolution. For iterative resolution, if a name server has the requested data in its cache, it returns the answer directly. Otherwise, it sends referral data to the client. The referral data are name server information for a domain that is closer to the authoritative domain with respect to the queried domain name. For recursive resolution, a name server attempts to find the answer for the query and then return it to the client.

There are four operations associated with a name server, namely $ReceiveQ_s$, $ReceiveR_s$, $SendQ_s$, and $SendR_s$. (The subscript s stands for "server".) They are for receiving queries, receiving responses, sending queries, and sending responses respectively. The protocol of the server can be described as follows:

```
Call Init_s with the root zone resource records, authoritative data, NS and A records
of child zones, and a set of trusted servers, if any.
for each DNS query received by calling ReceiveQ_s
    If performing recursive resolution
        while (query not resolved) {
            Call SendQ_s using s', a server "closer" to the authoritative zone;
            Call ReceiveR_s to receive the reply from s'
        }
        generate a reply based on View_s and those replies
    else /* iterative resolution */
        generates a reply that contains the answer (if known) and
        the NS and A records of s'
    Call SendR_s to reply to the querier
```

When a server s is initialized, it is given the NS RRs and the A RRs of a list of name servers for the root domain, denoted by ROOT. In BIND, name servers treat those RRs specially and do not remove them when their TTLs reach zero. The name server uses those hints to fetch the RRs of the name servers of the root domain and cache them. We combine these two steps in our model by putting RRs of the root name servers into $View_{cache}$. If a server is authoritative for a zone, say $z \in Zone$, it will store the zone data of z into its view. The zone data consists of the authoritative data for z, delegation data (i.e., the NS RRs for name servers managing its delegated zones cz, $\forall cz \in SubZone(z)$) and glue data (i.e., the A RRs for name servers managing zones cz, $\forall cz \in SubZone(z)$). The server s initializes the set of pending queries received, say $pending(s)$, and the set of

queries sent by itself, say $mypending(s)$, to $\emptyset$.

$$
\begin{aligned}
&Init_s(rootrrs : RR\text{-set},\ authrrs : RR\text{-set},\ childrrs : RR\text{-set},\ ss : Server\text{-set}) \\
&\mathsf{ext\ wr} : View_{auth}(s),\ View_{cache}(s),\ pending(s),\ mypending(s),\ TS(s) \\
&\mathsf{pre} :\ \mathsf{true} \\
&\mathsf{post} : View_{auth}(s) = \{((dname(arr), type(arr), class(arr)), ars) \mid \text{/*auth rr*/} \\
&\quad arr \in authrrs \wedge ars = \{rr \in authrrs \mid (dname(rr), type(rr), class(rr)) = \\
&\quad (dname(arr), type(arr), class(arr))\}\}\wedge \\
&\quad View_{cache}(s) = \{((dname(crr), \mathsf{NS}, class(crr)), crs) \mid \text{/*NS for child zones*/} \\
&\quad crr \in childrrs \wedge dname(crr) \in ZoneDelegated(s) \wedge type(crr) = \mathsf{NS}\wedge \\
&\quad crs = \{rr \in childrrs \mid (dname(rr), type(rr), class(rr)) = \\
&\quad (dname(crr), type(crr), class(crr))\}\}\dagger \\
&\quad \{((dname(crr_1), \mathsf{A}, class(crr_1)), crs) \mid crr_1,\ crr_2 \in childrrs \wedge\ \text{/*A for child zones*/} \\
&\quad type(crr_2) = \mathsf{NS} \wedge dname(crr_2) \in ZoneDelegated(s)\wedge \\
&\quad rdata(crr_2) = dname(crr_1) \wedge class(ccr_1) = class(ccr_2) \wedge type(crr_1) = \mathsf{A}\wedge \\
&\quad crs = \{rr \in childrrs \mid (dname(rr), type(rr), class(rr)) = \\
&\quad (dname(crr_1), \mathsf{A}, class(crr_1))\}\}\dagger \\
&\quad \{(\mathsf{ROOT}, \mathsf{NS}, class(rr)), rs) \mid rr \in rootrrs \wedge name(rr) = \mathsf{ROOT}\ \text{/*NS for root*/} \\
&\quad \wedge type(rr) = \mathsf{NS} \wedge rs = \{rr' \in rootrrs \mid (\mathsf{ROOT}, \mathsf{NS}, class(rr)) = \\
&\quad (dname(rr'), type(rr'), class(rr'))\}\}\dagger \\
&\quad \{(\mathsf{ROOT}, A, class(rr)), rs) \mid rr \in rootrrs \wedge name(rr) = \mathsf{ROOT}\ \text{/*A for root*/} \\
&\quad \wedge type(rr) = \mathsf{A} \wedge rs = \{rr' \in rootrrs \mid (\mathsf{ROOT}, \mathsf{A}, class(rr)) = \\
&\quad (dname(rr'), type(rr'), class(rr'))\}\}\wedge \\
&\quad pending(s) = \emptyset\wedge \\
&\quad mypending(s) = \emptyset\wedge \\
&\quad TS(s) = ss
\end{aligned}
$$

When a name server s receives a query, $ReceiveQ_s$ adds the query and the sender's identity into the state component $pending(s)$. When server s sends out a query, $SendQ_s$ adds the query and the identity of the queried server to the state component $mypending(s)$.

The post-condition of $ReceiveQ_s$ states that $pending(s)$ after the operation is equal to the union of $pending(s)$ before the operation and a singleton set consisting of a tuple representing the query received. For a state component x, $\overline{x}$ is used in the post-

condition predicate of an operation to denote the state for x before that operation.

$ReceiveQ_s(q : Query,\ from : Process)$
ext wr : $pending(s)$
pre : true
post : $pending(s) = \{(q, from)\} \cup \overline{pending(s)}$

$SendQ_s(q : Query,\ to : Server)$
ext wr : $mypending(s)$
pre : $(q, to) \notin mypending(s)$
post : $mypending(s) = \{(q, to)\} \cup \overline{mypending(s)}$

When server s receives a response from another server *from* that matches one of the queries s has sent to $from$, s calls $ReceiveR_s$ to accept the response. $ReceiveR_s$ then removes the corresponding entry from $mypending(s)$ and adds the DNS data in the response into its cache. These are described by the pre-condition and the post-condition of $ReceiveR_s$. The post-condition also states that the DNS data already in $View_{cache}(s)$ have precedence over those in the response message.

$ReceiveR_s(m : Resp,\ from : Server)$
ext wr : $mypending(s), View_{cache}(s)$
pre : $\exists (q, from) \in mypending(s) \cdot Q(q) = Q(m)$
post : $mypending(s) = \overline{mypending(s)} - \{(q, from) \mid q \in \overline{mypending(s)} \wedge Q(q) = Q(m)\}$
$\wedge\ View_{cache}(s) = \{(i \rightarrow rrs) \mid \exists rr_1 \in RRof(m) \cdot i = (dname(rr_1), type(rr_1), class(rr_1))$
$\wedge\ rrs = \{rr_2 \mid rr_2 \in RRof(m) \wedge i = (dname(rr_2), type(rr_2), class(rr_2))\}\}\dagger$
$Fresh(\overline{View_{cache}(s)})$

Given a query q and a destination *to*, server s calls $SendR_s$ to send a response for q. The post-condition of $SendR_s$ says that the resource records in the response are taken from $View(s)$, and s will remove (q, to) from $pending(s)$.

$SendR_s(q : Query,\ to : Process)\ m : Resp$
ext wr : $pending(s)$
ext rd : $View(s)$
pre : $(q, to) \in pending(s)$
post : $pending(s) = \overline{pending(s)} - (q, to) \wedge Q(m) = Q(q) \wedge \forall rr \in RRof(m) \cdot$
$(\exists i \in Fresh(View(s)) \wedge i = (dname(rr), type(rr), class(rr)) \wedge rr \in Fresh(View(s))i)$

4.7.5 Assumptions

In this section, we explicitly list our assumptions for DNS. They concern with how name servers prioritize RR sets, the accuracy of authoritative DNS data, the effect of changes on DNS data, the accuracy of delegation data, and the power of attackers on eavesdropping DNS packets.

Assumption 10 *Protected servers do not add an RR to the* $View_{cache}$ *of a process if an RR that corresponds to the same index already exists in the* $View_{cache}$*. Moreover, protected servers prefer authoritative data over cache data.*

Both of them hold for "good" servers (i.e., servers that behave according to the DNS RFC [43, 44]). Some server implementations rank data from different sources at different credibility levels. Moreover, data from a higher credibility level can preempt data from a lower credibility level. We do not model data credibility levels in our work for the sake of simplicity. As we will see later, because we only allow authoritative data to reach a protected name server, this simplification does not affect the validity of our results.

Assumption 11 *Data from an authoritative server are correct.*

For example, if a server is authoritative for a machine h and the server says the IP address of h is i, then we believe that the IP address of h is i.

Assumption 12 *When a server attaches a TTL with* t *seconds to a resource record for which the server is authoritative, the resource record will be valid for the next* t *seconds.*

We state this assumption because there is no revocation mechanism in DNS. Without this assumption, one cannot determine the validity of DNS data as soon as they leave their authoritative servers. We argue that this assumption is reasonable. When a resource record needs to be changed, the TTL of this resource record is usually decreased before the changeover so that incorrect/stale records will timeout shortly after the changeover.

Assumption 13 *For every zone, the delegation data and the glue data of its child zones correspond to the* NS *RRs and the* A *RRs of the name servers of the child zones (i.e., no* lame delegation*).*

Lame delegation is caused by operational errors: A system administrator changes the name servers for a zone without changing the corresponding RRs in the parent zone or notifying the system administrator of the parent zone about the change.

Assumption 14 *Attackers cannot eavesdrop on the DNS packets sent between our protected servers and the legitimate name servers.*

This is a limit we place on the attackers; if attackers can monitor the communication, our scheme may fail to cope with spoofing attacks. In the future, when the use of DNSSEC is widespread, we may drop this assumption and depend on DNSSEC to authenticate messages. An implication of this assumption is that by randomizing the query id used, the probability that an attacker can forge a response whose id matches the randomized query id is negligible. Thus attempts for sending forged responses by guessing the query id used can be detected by the wrapper.

4.7.6 Our Goal

Our goal is to ensure that the view of a protected name server agrees with those of the corresponding authoritative name servers. This goal is specified using a VDM data invariant. A *data invariant* of a data type specifies the predicates that must hold true during the execution of a system. Our name server specification (c.f. Section 4.7.4), which reflects the minimal functionalities of DNS servers among existing implementations, does not satisfy this data invariant because it allows non-authoritative DNS data to reach a name server. Thus for a name server s, $Authoritative(\mathsf{rng}\ View(s))$ may not hold. In the next section, we will present our solution—a security wrapper for protecting name servers. Our DNS wrapper filters out DNS messages containing resource records that cannot be verified as authoritative. Therefore, a protected name server that satisfies the data invariant can be constructed by composing a name server and our DNS wrapper.

$$
\begin{array}{l}
\mathsf{state}\ DNS\ \mathsf{of} \\
\quad protectedNS : Server\text{-}\mathsf{set} \\
\ldots \\
\mathsf{inv}\ mk\text{-}DNS(protectedNS) \triangleq \\
\quad \forall s \in protectedNS \cdot Authoritative(\mathsf{rng}\ View(s)) \\
\mathsf{end}
\end{array}
$$

4.8 Our DNS Wrapper

We use *security wrapper* (or simply wrapper) to refer to a piece of software that encapsulates a component, such as a name server, to improve its security. Using wrappers to enhance the security of existing software is not a new idea. Related works include TCP wrapper [69], and generic software wrappers [23]. However, our work is different in that it addresses problems that are DNS specific and it involves the use of formal specifications.

We considered two different retrofit approaches to secure DNS. The first approach is a non-intrusive, conventional network intrusion detection approach. In this approach, a network sniffer may be used to *monitor* the DNS traffic flowing into and out of a name server, and to report suspicious DNS messages. The second approach uses a name server wrapper that functions as a "smart" filter: All messages sent between a protected name server and the rest of the world pass through the wrapper. Moreover, those that contain suspicious DNS resource records are discarded and recorded in a log file. Section 4.8.1 justifies our choice of using a filter-based approach.

Consider a wrapper w. Wrapper w checks DNS response packets going to a name server and ensures that they are authenticated[29] and they agree with authoritative answers. If a resource record in the response does not come from an authoritative server, wrapper w locates an authoritative server and queries that server for the authoritative answer. To locate an authoritative server for a zone, say z, the wrapper starts with a server, say s, that is known to be an authoritative server for an ancestor zone of z, and queries server s for authoritative servers of the child zone that is either an ancestor zone of z or z itself. The search is performed by traversing the domain name tree, one zone at a time, until an authoritative server for the DNS data being verified is located. Recall that the zone data maintained by a server include the name server data of the delegated zones. Moreover, in BIND, those data are rated at the highest credibility level. (We emulate this behavior by initializing $View_{cache}$ with the delegated zone data at startup time.) Thus those data will not be overridden by incorrect resource records sent by an attacker. In other words, those name server data (i.e., zone cut data and glue data) are immune from cache-poisoning attacks. Our scheme exploits this fact to securely locate the authoritative servers.

[29] Data authentication checks can be performed by matching the query *id*'s of queries to those of responses, or by using DNSSEC. However, the query *id* generation process used in some implementations of name servers is quite predictable. Before DNSSEC is widely deployed, we need a means to protect these name servers from spoofing attacks.

4.8.1 Wrapper-based Design Versus Sniffer-based Design

For the following reasons, we selected a wrapper-based design rather than a sniffer-based design, which is commonly used in network intrusion detection work, to protect name servers.

- *The sequence of DNS queries and responses destined for a name server observed by the wrapper is the same as that observed by the name server*: A sniffer-based monitor may suffer from problems like packet loss and packet reordering, causing it to miss attacks. Packet loss is severe when a sniffer is overwhelmed by network traffic. A sniffer and a name server may observe different packet orderings, say, when packets are reordered by a router that is situated between them. If multiple responses for a query (which may include forged responses) are received by a name server, in certain server implementations, the first one will be used and the rest will be discarded. Thus if the sniffer and the server may observe different packet orderings, the sniffer may not know which of these responses will be used by the server. Running a multi-homed name server[30] exacerbates these problems: DNS messages for the multi-homed name server may arrive from different networks. Thus sniffers that monitor these networks have to coordinate among themselves to reconstruct the trace of DNS messages observed by the name server.

- *Security improvements such as network traffic encryption and DNS access control may impede the ability of a sniffer to interpret DNS messages destined for a name server and to obtain authoritative answers for wrapper verification*: When IPSEC (IP Security Protocol) [30] becomes widely used, DNS traffic between a name server and other hosts or firewalls may be encrypted. Using a network sniffer may no longer be feasible or desirable in this case because of the additional overhead of key management and the potential reduction in security for distributing the secret keys to the sniffer and protecting them. Access control mechanisms have been incorporated into some DNS implementations. Specifically, a name server may be configured to accept requests only from certain hosts. Thus a request sent by another machine (as often the case in a sniffer-based design) may be ignored if that machine does not have the permission to access the name server. In other words, the sniffer-based monitor

[30] A multi-homed name server runs on a machine that is connected to two or more networks.

might be unable to contact name servers authoritative for certain domain names to obtain authoritative answers.

- *A wrapper can randomize the query id's used by a name server to cope with some spoofing attacks*: The wrapper design allows us to increase the security of query *id* generation without modifying the name servers. In some DNS implementations, the query *id*'s of successive queries generated by a name server always differ by one. In other words, the query *id*'s are quite predictable. Before DNSSEC is widely deployed, using the query *id* is one of the major means to detect counterfeit DNS response messages. To masquerade as a particular server, an attacker needs to send a DNS response whose query *id* is the same as that of the corresponding query. Unless the attacker can monitor the DNS traffic between our name server and the legitimate name server, randomizing the query *id* can be quite effective for detecting those counterfeit response messages. In our design, the wrapper replaces the query *id* with a randomly generated query *id* (that differs with those of outstanding queries) and maintains a translation table between the *id* used by the name server and the *id* used in the packets sent to the network. When a response is received, the wrapper checks the translation table for a matching query *id*. If a match is found, the query *id* of the response will be translated to that used by the protected server before the response is forwarded. Otherwise, the response is dropped and put into the security log. This query *id* translation technique cannot be used in a passive monitoring scheme.

- *Ptacek's and Newsham's [53] paper discussed some common vulnerabilities for sniffer-based intrusion detection systems. Those vulnerabilities are not applicable to wrapper-based systems*: In [53], Ptacek and Newsham described techniques to defeat sniffer-based intrusion detection systems, including denial of service attacks against a sniffer, and insertion and evasion tricks to escape detection. For example, an attacker may construct an IP packet with a calculated time-to-live value such that the sniffer-based network monitor will accept the packet, but the destination host will not. Another example is that an attacker may exploit the differences between a sniffer and a host in handling packet fragmentation and packet reassembly: The attacker can construct a packet fragment that is ignored by the sniffer-based network monitor, but is accepted by the destination host. Attacks like these are possible because of the decoupling between the sniffer and the protected hosts. Employing a wrapper that

runs on the name server machine can either make the system "fail-close" (as opposed to "fail-open") or prevent those potential vulnerabilities. A protection mechanism is *fail-close* if compromising its availability also makes the protected system unavailable.

4.8.2 Overview of the Specification of Our DNS Wrapper

To enforce our security goal defined in Section 4.7.6, we characterize (in VDM) a DNS wrapper that prevents a name server from using DNS data that disagree with the corresponding authoritative answers.

Let w denote our DNS wrapper and ns denote the name server protected by w. Our wrapper consists of two main parts: $Wrapper_s q$ for processing queries, and $Wrapper_s r$ for processing responses. (The subscript s stands for "server".) Wrapper w processes queries generated by ns before they are sent out, and processes queries destined for ns. Wrapper w also processes responses destined for ns; those that are accepted by w will be forwarded to ns.

When ns sends a query, wrapper w generates a random query id and uses it to replace the original query id (used by ns). We use a translation table to track the mapping between the random query id's used by w and the original query id's used by ns. We use a map *transTable:* $QID \times QuestionSec \rightarrow QID$, initialized to null, to implement the translation table, where *QID* is the type for a query id, and *QuestionSec* is the type for the question section of queries. Using *QID* alone in the domain of *transTable* is insufficient because w may err in translating query id's when ns reuses its query id's. Thus the domain of *transTable* contains both the original query id and the question section of a query.

Function *randomId:* $QID-\mathsf{set} \rightarrow QID$ returns a random query id that does not appear in the input query id set. Wrapper w uses this function to randomize the id's of the queries generated by ns and to generate random query id's for its own queries.

Function *wfq: Query* $\rightarrow$ *Boolean* returns true if the input query is well-formed; otherwise, it returns false. Recall that there are two types of queries: standard and inverse. A standard query looks for resource records given a domain name. Given a standard query, *wfq* checks the question section of the query to determine whether it is well-formed. An inverse query looks for the domain name corresponding to a given resource record. Given an inverse query, *wfq* checks the resource record in the answer section for well-formedness.

$transTable : QID \times QuestionSec \rightarrow QID$

$randomId : QID\text{−}\mathsf{set} \rightarrow QID$

/* randomId(s) randomly picks a query id that does not appear in the QID-set s */

$wfq : Query \rightarrow Boolean$ /* well-formed queries */

$\forall q \in Query \cdot wfq(q) \triangleq$

if opcode(q) = STANDARD-QUERY

/* standard queries look for resource data given a domain name */

$wfqs(Q(q)) \wedge Ans(q) = \mathsf{null} \wedge Auth(q) = \mathsf{null} \wedge Add(q) = \mathsf{null}$

elseif opcode(q) = INVERSE-QUERY

/* inverse queries look for a domain name given resource data */

$Q(q) = \mathsf{null} \wedge wfias(Ans(q)) \wedge Auth(q) = \mathsf{null} \wedge Add(q) = \mathsf{null}$

$wfqs : QuestionSec \rightarrow Boolean$ /* well-formed question section */

/* returns true if the question section consists of a valid domain name,
a valid query type, and a valid query class; otherwise returns false */

$wfias : AnswerSec \rightarrow Boolean$ /* well-formed answer section for inverse queries */

/* returns true if the answer section consists of only one valid resource record
(of type A usually); otherwise returns false */

4.8.3 The Specification of Operation $Wrapper_s q$

$Wrapper_s q$ processes queries that involve ns. These queries can be partitioned into two types. The first type corresponds to the queries that are sent to ns. The second type corresponds to the queries that are generated by ns. These two types of queries are treated differently. For the first type, wrapper w checks the queries to determine whether they are well-formed (e.g., the answer, the authority, and the additional sections for a standard query should be empty). For the second type, the wrapper generates a random query id, replaces the query id used in the original query by this randomly generated query id, and updates the local query id translation table.

Given a query q sent from ns to a process to, $Wrapper_s q$ returns p (which is the same as q except the query id of q is replaced by a randomized query id), and updates the translation table *transTable* if necessary. For queries sent by processes other than ns (i.e., those sent by resolvers destined for ns), $Wrapper_s q$ returns q if the query is well-formed; otherwise, it returns null.

Recall that $Q(m)$ denotes the question section of message m, and $id(m)$ denotes the query id of message m. Let $HdrMinusId(m)$ denote the header section minus the id field of message m.

$Wrapper_s q(q : Query,\ to : Process,\ from : Process)\ p : Query$

ext wr : $transTable$

pre : true

post : $(from \neq ns \wedge wfq(q) \Rightarrow p = q) \wedge$

$(from \neq ns \wedge \text{not } wfq(q) \Rightarrow p = \text{null}) \wedge$

$(from = ns \wedge (id(q), Q(q)) \notin \text{dom } \overline{transTable} \Rightarrow$

/* Our name server sends out a new query */

$(id(p) = randomId(\text{rng } \overline{transTable}) \wedge HdrMinusId(p) = HdrMinusId(q) \wedge$

$Q(p) = Q(q) \wedge transTable = \overline{transTable} \dagger \{(id(q), Q(q)) \rightarrow id(p)\}) \wedge$

$(from = ns \wedge (id(q), Q(q)) \in \text{dom } \overline{transTable} \Rightarrow$

/* Our name server re-sends a pending query due to timeouts. The wrapper uses transTable to modify the query id so that all query messages corresponding to the same query will have the same query id */

$id(p) = transTable(id(q), Q(q)) \wedge HdrMinusId(p) = HdrMinusId(q) \wedge$

$(Q(p) = Q(q) \wedge transTable = \overline{transTable})$

4.8.4 The Specification of Operation $Wrapper_s r$

$Wrapper_s r$ processes responses that involve ns. $Wrapper_s r$ has two components: $Wrapper_s r1$ and $Wrapper_s r2$. $Wrapper_s r1$ screens out forged response messages. $Wrapper_s r2$ verifies the response messages to ensure that they agree with authoritative answers. There are two types of responses received by a wrapper: responses for queries generated by the protected name server ns, and responses for queries generated by the wrapper itself (for wrapper verification purposes). When a response for a query generated by ns is received, the wrapper uses the query id translation table to restore the query id (to the one used by ns) before passing the response to $Wrapper_s r2$.

Wrapper w uses a state component *completed: Query* $\times$ *TimeStamp* $\times$ *QID* to keep track of the queries that have been responded. The *TimeStamp* field records the time of receiving the first response. For queries generated by the protected name server, the *QID* field records the randomized query id generated by the wrapper that replaces the query id used by the protected name server. For queries generated by the wrapper itself,

randomized query id's are used and both the query id and the *QID* fields in *completed* have the same value.

4.8.5 The Specification of Operation $Wrapper_s r1$

Given a response message m sent by a process *from* to a process *to*, $Wrapper_s r1$ returns m (after converting the query id back to that used by the server if m corresponds to a server-generated query) in the following cases:

- There is a matching query for m (either generated by the protected name server ns or wrapper w itself).
- Response m corresponds to a query that has been responded within the past "threshold" seconds (explained below).

Otherwise, $Wrapper_s r1$ returns null. In other words, when the wrapper observes a forged response with an incorrectly guessed query id or a forged response that does not correspond to any recent queries, $Wrapper_s r1$ returns null. Sometimes a name server/wrapper sends out multiple copies of a query (possibly to more than one name servers), which happens when the internal timer expires before receiving a response. Thus multiple response messages for the same query may be received. We keep matching query-response pairs in the wrapper for at least *threshold* seconds, and use them to cope with these multiple responses.

The map deletion operator, denoted by $\ntriangleleft$, takes two operands. The first operand is a set and the second one is a map. The operator gives a map that contains all elements in the second operand whose domains do not belong to the first operand. Let now denote the current time, and $MsgMinusId(m)$ denote message m minus the query id field.

$Wrapper_s r1(m : Resp)\ p : Resp$

ext wr : $completed,\ transTable$

pre : true

post : $(\exists q \in Query \cdot$

/* When m is a response to a pending query generated by the protected server */

$(id(q), Q(q)) \in \mathsf{dom}\ transTable \wedge transTable(id(q), Q(q)) = id(m)\ \wedge$

$Q(m) = Q(q) \Rightarrow$

$((id(q), Q(q)) \lessdot transTable \wedge completed = \overline{completed} \cup (q, \mathsf{now}, id(m))\ \wedge$

$id(p) = id(q) \wedge MsgMinusId(p) = MsgMinusId(m)))\ \wedge$

$(\exists q \in Query,\ ts \in TimeStamp,\ i \in QID \cdot$

/* When m is associated with a query that has been responded already */

$Q(m) = Q(q) \wedge id(m) = i\ \ \wedge (q,\ ts,\ i) \in \overline{completed} \wedge ts > (\mathsf{now} - threshold)\ \Rightarrow$

$id(p) = id(q) \wedge MsgMinusId(p) = MsgMinusId(m))\ \wedge$

$(\nexists q \in Query,\ ts \in TimeStamp,\ i \in QID \cdot$

$((id(q), Q(q)) \in \mathsf{dom}\ transTable \wedge transTable(id(q), Q(q)) = id(m)\ \wedge$

$Q(m) = Q(q))\ \vee$

$(Q(m) = Q(q) \wedge id(m) = i\ \ \wedge (q,\ ts,\ i) \in \overline{completed} \wedge ts > (\mathsf{now} - threshold))\ \Rightarrow$

$p = \mathsf{null})$

4.8.6 The Specification of Operation $Wrapper_s r2$

If a response m passes $Wrapper_s r1$, the wrapper invokes $Wrapper_s r2$ to verify the resource records of m. If all resource records in m agree with authoritative answers, $Wrapper_s r2$ returns true and updates $View_{cache}(w)$ with those resource records. Otherwise, $Wrapper_s r2$ returns false.

```
Wrapper_s r2(m : Resp) violation : Boolean
ext wr : View_cache(w)
var : addset : DbMap /* rr in a message for a (domain name class, type) must not be
        mixed with rr for the same (domain name, class, type) from another message */
  violation = false;
  addset = ∅;
  foreach rr ∈ RRof(m) {
      if not AuthVerified(rr,from,addset) then violation = true;
      if (dname(rr), type(rr), class(rr)) ∈ dom(addset) /* add rr to addset */
      then addset(dname(rr),type(rr),class(rr)) =
          addset(dname(rr),type(rr),class(rr)) ∪ {rr};
      else addset(dname(rr),type(rr),class(rr)) = {rr};
  }
  if not violation ∧ not tc(m) /* must never store incomplete record sets */
  then View_cache(w) = addset † Fresh(overline{View_cache(w)})
  /* addset contains zone data from auth zones */
pre : m ≠ null ∧ Authoritative(rng View(w))
post : (Authoritative(RRof(m)) ⇒ violation = false ∨
        not(Authoritative(RRof(m))) ⇒ violation = true) ∧
        Authoritative(rng View(w))
```

4.8.7 The Specification of Operation *AuthVerified*

$Wrapper_s r2$ uses *AuthVerified* to verify a single resource record. Given a resource record rr sent by a process *from*, and a *DbMap* containing accepted resource records that belong to the same response message as rr does, *AuthVerified* returns true if rr is known by the wrapper to be authoritative, or rr agrees with an authoritative answer. Otherwise, *AuthVerified* returns false. There are three cases in which rr is known to be authoritative: (1) *from* is known to be an authoritative server for rr; (2) *from* is known to be an authoritative server for a zone and rr corresponds to delegation information for the server's delegated zones; (3) rr can be found in $View(w)$. If none of the above three conditions is met, the wrapper contacts an authoritative server for rr to verify rr against the authoritative answer.

Let $\bigcup$ denote the distributed union operator (i.e., $\bigcup$ forms the union of all sets inside a set). The function $ZtoD(z)$ returns the domain name corresponding to a zone z.

AuthVerified calls *KnownAuthServer*, *NearestZoneKnown*, and *CheckAuthServer*. *KnownAuthServer* returns a set of authoritative name servers for a specified domain name. Given a process p and a resource record, *NearestZoneKnown* returns the nearest zone with respect to that resource record for which p knows an authoritative server (i.e., the authoritative server data of that zone is in $View(p)$). *CheckAuthServer* contacts authoritative servers with respect to a specified resource record and returns the authoritative answer. For the sake of clarify, the specifications of *KnownAuthServer* and *NearestZoneKnown* are shown in Appendix B. The specification of *CheckAuthServer* will be shown after that of *AuthVerified*.

AuthVerified($rr : RR,\ from : Process,\ knownNS : DbMap$) $auth : Boolean$
  /* returns true if rr agrees with authoritative data from an authoritative server; returns false otherwise. knownNS contains verified NS resource records in the same response as rr does. */
ext wr : $View(w)$
var : $ans:\ Server,\ q:\ Query$
  auth = false;
  if ((from $\in KnownAuthServer(dname(rr), class(rr), Fresh(\overline{View(w)})))\ \vee$
    /* *from* is an auth server */
    ($\exists$ z $\in$ Zone $\cdot$ from $\in KnownAuthServer(ZtoD(z), class(rr), Fresh(\overline{View(w)}))$
      /* *from* gives info of delegated zones */
      $\wedge$ ((dname(rr) $\in$ SubZone(z) $\wedge$ type(rr) = NS) $\vee$
      ($\exists$ cz $\in$ Zone, nsr $\in \bigcup$(rng knownNS) $\cdot$ cz $\in$ SubZone(z) $\wedge$ type(rr) = A $\wedge$ dname(nsr) = cz $\wedge$ type(nsr) = NS $\wedge$ dname(rr) = rdata(nsr) $\wedge$ dname(rr) $\in$ SubDomain(z)))) $\vee$ /* *dname(rr)* is under *z*'s domain */
    (rr $\in \bigcup$(rng $Fresh(\overline{View(w)})$))) /**rr* is known to be auth */
  then auth = true
  else
    nz = *NearestZoneKnown*(w,rr);
    /* find nearest zone that w knows its name servers */
    auth = rr $\in$ Ans(CheckAuthServer(rr,ZtoD(nz),nz,1))
pre : $Authoritative$(rng $View(w)$)
post : $auth = Authoritative(\{rr\}) \wedge Authoritative$(rng $View(w)$)

4.8.8 The Specification of Operation *CheckAuthServer*

When *AuthVerified* cannot verify a resource record *rr* using its local information, *AuthVerified* invokes *CheckAuthServer* to query an authoritative server for *rr* to obtain an authoritative answer and compares *rr* with the authoritative answer to check for consistency. *CheckAuthServer* has two main steps. First, *CheckAuthServer* locates an authoritative name server for *rr* by traversing the domain name space one zone at a time, starting from a given zone *nzone*. After an authoritative servers for *rr* are located, *CheckAuthServer* will query this server to obtain an authoritative answer with respect to *rr*.

Given a resource record *rr*, *CheckAuthServer* returns the authoritative answer for *(dname(rr),type(rr),class(rr))*, if found; otherwise, *CheckAuthServer* returns null. To prevent reference loops caused by misconfigurations or possible denial of service attacks, we use a counter *hop* to record the amount of resources used by *CheckAuthServer*. If the counter exceeds a certain threshold, *HopThreshold*, *CheckAuthServer* terminates and returns null.

CheckAuthServer uses $ConstructQ_w$, Get_w, *IsAuthDomain*, and GETNS-A. Their details are shown in Appendix B. Given a domain name, a resource type, and a resource class, $ConstructQ_w$ constructs a query with a randomized query id. Get_w sends a query to a specified name server and returns the response, if any. Given a resource record *rr* and a domain name *d*, *IsAuthDomain* returns true if *d* is the authoritative domain for *rr*; otherwise, it returns false. The macro GETNS-A retrieves the NS resource records and the A resource records for a specified domain. The specification of *CheckAuthServer* is as follows:

```
CheckAuthServer(rr : RR, nzone : Zone, curdom : DName, hop : Integer, rs : DbMap)
   r : Reply
ext rd : View(w)
var : curr-ns : Server, ass : Server−set, q : Query, m : Reply, newcurdom : DName,
   nrr, arr, temprr, temprr2 : RR, tempzone : Zone, temprs : RR−set,
   nss : RR−set, newrs : DbMap
   if hop ≥ HopThreshold then return null
   newrs = rs
   m = null
   ass = KnownAuthServer(ZtoD(nzone),class(rr),newrs † Fresh(View(w)))
   if IsAuthDomain(rr,curdom) then  /* curr-ns is an auth server for rr */
       if (rng transTable) = QID then return null;  /* unused query id available */
       q = ConstructQ_w(dname(rr), type(rr), class(rr)); /* make query q for rr */
       while m = null ∧ ass ≠ ∅ do /* enumerate auth servers of zone nzone */
           let curr-ns ∈ ass in
               m = Get_w(q,curr-ns)
               ass = ass - {curr-ns}
       return m;
   else  /* find auth servers for rr */
       newcurdom = ChildDomain(nzone,rr);
       /* find the child domain of curdom that is closer to the auth zone for rr */
       if (rng transTable) = QID then return null;
       q = ConstructQ_w(newcurdom, NS, class(rr)); /* find NS rr for newcurdom */
       while m = null ∧ ass ≠ ∅ do
           let curr-ns ∈ ass in
               m = Get_w(q,curr-ns)
               ass = ass - {curr-ns}
       if m = null then return null
       if rcode(m) = NotExists then /* newcurdom not a zone */
           return CheckAuthServer(rr,nzone,newcurdom,hop,newrs)
       else GETNS-A  /* get NS and A rr for newcurdom servers */
    if KnownAuthServer(newcurdom,class(rr),newrs † Fresh(View(w))) ≠ ∅
        /* exists a server for newcurdom that we know the A rr */
    then return CheckAuthServer (rr,newcurdom,newcurdom,hop+1,newrs)
    else return null /* can't locate server for newcurdom */
pre : Authoritative(rng View(w))
post : Authoritative(Ans(r))
```

4.9 Experiments

4.9.1 Overview

We conducted experiments to evaluate the response time (i.e., the elapsed time between sending a query to a name server and receiving a response from it) of a wrapped name server, and to evaluate the false positive rate, the false negative rate, and the computational overhead (i.e., CPU time used) of our wrapper.

Based on the DNS wrapper specification discussed in Section 4.8, we implemented a prototype of the DNS wrapper for BIND release 4.9.5, which was the latest release for BIND when we started our implementation. The DNS wrapper was written in C. We modified the BIND name server source code to invoke the DNS wrapper upon receiving queries and responses and upon sending queries to other name servers.

In this section, we describe three sets of experiments and their results. In Experiment #1, we examined the response time and the false positive rate of our wrapper using 100 domain names that were distinct and remote (i.e., outside the cs.ucdavis.edu domain). In Experiment #2, we examined the response time, the false positive rate, and the computational overhead of our wrapper using a trace of DNS queries received by a name server in an operational setting. In Experiment #3, we examined the false negative rate of our wrapper with respect to four attacks: three cache poisoning attacks and one masquerading attack.

4.9.2 General Experimental Setup

In these experiments, our name servers listened to port 4000 instead of port 53 (the *de facto* standard port number for name servers) for DNS queries to prevent queries outside our experiments from affecting our results.

In every run of our experiments, we started a fresh copy of our name server because name servers maintain a cache for DNS information obtained through interacting with other name servers. The behavior of a name server can be quite different depending on whether the DNS information queried is in the cache. Restarting name servers can avoid interference between consecutive runs of the experiment.

We used a modified version of *nslookup* as the DNS client in our experiments. (See [1] for a good tutorial on *nslookup*.) We chose *nslookup* because it is a convenient

tool for generating DNS queries and displaying DNS responses. Moreover, *nslookup* can be easily configured to use a specified name server port number and to query a specified name server. Our modified *nslookup* uses *gethrtime*() Unix system calls to record the time when a query is sent and when the corresponding response is received. Unless otherwise specified, we use *nslookup* to refer to this modified version of *nslookup* for the rest of this chapter.

Our experiments were performed on a lightly loaded Sun SPARC-5 running Solaris 2.5.1. We ran our name servers and *nslookup* on the same machine (instead of running them on separate machines) to eliminate the network latency for the communication between *nslookup* and our name servers. Thus we reduced the influence of the local area network load on the experimental results.

Because we did not have control over external name servers, and the inter-network links between our name server and external name servers, we performed Experiments #1 and #2 multiple times and calculated the average response time.

4.9.3 Experiment #1

Experiment #1 was designed to evaluate the response time of a wrapped name server and the false positive rate of our wrapper given a list of domain names that were distinct and remote (i.e., outside our authoritative domain).

Data Set

The data set for Experiment #1 consisted of the domain names of a collection of 100 web sites. These web sites were selected by PC magazine online as the "Top 100 web sites" in 1998 [49]. They were classified into 20 categories—for example, online auctions, shopping, news, entertainment, search engines—covering different aspects of Internet usage. (For the sake of completeness, the 100 domain names are listed in Appendix C.) We used this list instead of a random list of domain names because the domain names in the selected list generally corresponded to popular and useful web sites. Thus the selected list probably provided us a more representative sample set of remote domain names than a random list would have. This data set represents a worst-case scenario for the response time overhead for the wrapper. In practice, the stream of queries received by a name server usually contains some repetitions and "local" queries (i.e., queries for domain names for which

the name server is authoritative). Both repetitions and local queries reduce the amount of wrapper verification needed. (In Section 4.9.4, we discuss the performance of the DNS wrapper in a "real life" setting.)

Experimental Procedure

1. Start a wrapped name server.
2. Run *nslookup* to query the wrapped name server for the IP addresses of the 100 domain names sequentially.
3. Terminate the wrapped name server.
4. Start an unmodified name server.
5. Run *nslookup* to query the unmodified name server for the IP addresses of the 100 domain names sequentially.
6. Terminate the unmodified name server.

Experimental Results

Table 4.2 shows the statistics related to response times for *nslookup* based on 48

	Unmodified Name Server				Wrapped Name Server			
# queries	Mean	Min	Max	Std Dev	Mean	Min	Max	Std Dev
20	16.01	3.37	71.05	10.84	30.94	8.35	67.86	11.77
40	22.69	9.99	77.78	11.35	44.47	15.06	84.69	14.15
60	31.61	15.98	91.83	12.66	58.27	30.99	99.16	16.07
80	36.98	18.45	102.37	13.48	67.37	40.39	116.59	17.52
100	47.91	30.29	112.70	13.08	85.46	51.01	133.61	18.78

Table 4.2: Cumulative Response Time (in Sec.) for the "Top 100 Web Sites" Data Set.

runs of the experiment. A value in the "# queries" column refers to the cumulative number of queries starting from the first query. For example, the "60" in that column indicates that the corresponding row provides statistics for cumulative response times for the first 60 domain names in the list. The "Mean", "Min", "Max", and "Std Dev" columns indicate the average, the minimum, the maximum, and the standard deviation of the cumulative

response times among the runs. The response times for the wrapped name server were larger than the counterparts for the unmodified name server, and the differences represented the overheads of the wrapper. For 100 queries, the mean response time for the wrapped server was 0.8546 second per query, and that for the unmodified server was 0.4791 second per query.

To find out the false positive rate of our wrapper, we examined the security violation messages in the security log generated by the wrapper, and investigated the validity of these violations (manually) by querying the relevant name servers. Among the 48 runs of the experiment, the average number of security violations that corresponded to different domain names was 2.125.

We define two classes for the false positives generated by our wrapper. The first class, called *pseudo false positive* (or PFP in short), corresponds to name server behaviors that violate our name server specification or to a violation of our assumptions. For example, false positives caused by misconfigurations of name servers are in PFP. An example of misconfiguration is *lame delegation.* In order for domain delegation to work, the name servers of a (parent) zone must know the name server information of its child zones—the domain names of the child zone name servers and, if these domain names belong to the parent domain, the IP addresses of these child zone name servers. In lame delegation, the parent zone has incorrect information about the name servers of its child zones. Lame delegation may be caused by an operational error in which a system administrator changes the name server data of a zone but fails to have the corresponding data in the parent zone updated.

To illustrate anomalies for class PFP, Figure 4.8 shows a security violation message reported by our wrapper. In this example, our wrapper received a response for querying the IP address of www.zone.com. Note that in the response (as shown in the upper part of Figure 4.8) there are two "additional" records that contain the IP addresses of dns1.moswest.msn.net. and dns2.moswest.msn.net. The wrapper verified those IP addresses with an authoritative server for those two domain names. From the response sent by the authoritative server, shown in the lower part of Figure 4.8, the wrapper discovered that they did not exist and thus generated a security log message.

A second example for class PFP is illustrated in Figure 4.9. It represents an operational error that DNS data are associated with an alias rather than with the canonical domain name. This is a violation of the DNS specification [43, 44] that forbids CNAME

```
seclog: type = 10, caller = CheckAuthServer2: .
Unable to verify a rr.
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52426
;; flags: qr aa rd ra; Ques: 1, Ans: 6, Auth: 3, Addit: 2
;; QUESTIONS:
;;      www.zone.com, type = A, class = IN

;; ANSWERS:
www.zone.com.   A       207.46.172.41
www.zone.com.   A       207.46.172.42
www.zone.com.   A       207.46.172.43
www.zone.com.   A       207.46.172.44
www.zone.com.   A       207.46.172.45
www.zone.com.   A       207.46.172.46

;; AUTHORITY RECORDS:
zone.com.       NS      atbd.microsoft.com.
zone.com.       NS      dns1.moswest.msn.net.
zone.com.       NS      dns2.moswest.msn.net.

;; ADDITIONAL RECORDS:
dns1.moswest.msn.net.   A       204.255.246.17
dns2.moswest.msn.net.   A       204.255.246.18

------------------------------------------------------------
seclog: type = 12, caller = CheckAuthServer2: .
Supplementary Info.
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 19455
;; flags: qr aa ra; Ques: 1, Ans: 0, Auth: 1, Addit: 0
;; QUESTIONS:
;;      dns1.moswest.msn.net, type = A, class = IN

;; AUTHORITY RECORDS:
msn.net.        SOA     dns.cp.msft.net. msnhst.microsoft.com. (
                        990318007       ; serial
                        21600   ; refresh (6 hours)
                        3600    ; retry (1 hour)
                        1728000 ; expire (20 days)
                        21600 ) ; minimum (6 hours)
```

Figure 4.8: A Security Log Message for www.zone.com.

```
seclog: type = 10, caller = CheckAuthServer2: .
Unable to verify a rr.
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 64463
;; flags: qr aa rd ra; Ques: 1, Ans: 1, Auth: 6, Addit: 5
;; QUESTIONS:
;; www.3com.com, type = A, class = IN

;; ANSWERS:
www.3com.com.   A       192.156.136.22

;; AUTHORITY RECORDS:
3com.com.       NS      four11.3com.com.
3com.com.       NS      tmc.edu.
3com.com.       NS      ns1.cs.odu.edu.
3com.com.       NS      news.aero.org.
3com.com.       NS      mail.aero.org.
3com.com.       NS      seaweed.thirdcoast.net.

;; ADDITIONAL RECORDS:
four11.3com.com.        A       129.213.128.98
tmc.edu.                A       128.249.1.1
ns1.cs.odu.edu.         A       128.82.4.38
news.aero.org.          A       130.221.16.4
seaweed.thirdcoast.net. A       207.38.56.10

------------------------------------------------------------
seclog: type = 12, caller = CheckAuthServer2: .
Supplementary Info.
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 33450
;; flags: qr aa ra; Ques: 1, Ans: 2, Auth: 3, Addit: 1
;; QUESTIONS:
;; ns1.cs.odu.edu, type = A, class = IN

;; ANSWERS:
ns1.cs.odu.edu.         CNAME   nixon.cs.odu.edu.
nixon.cs.odu.edu.       A       128.82.4.38
[snipped]
```

Figure 4.9: A Security Log Message for www.3com.com.

data and another type of data for the same domain name to co-exist. In the experiment, our wrapper received a response, as shown in the upper part of Figure 4.9, that included an A resource record (i.e., the IP address) for ns1.cs.odu.edu. The wrapper verified it with the authoritative server for ns1.cs.odu.edu. and discovered that this domain name did not have an A resource record (c.f. the lower part of Figure 4.9).

The second class for the false positives, called *genuine false positive* (or GFP in short), is for warnings generated by the DNS wrapper that do not correspond to violations of our specification or of our assumptions. For the class GFP, all but one of the security log messages generated by the wrapper in Experiment #1 were due to the number of DNS queries sent in the wrapper verification exceeding a threshold. The threshold was used to ensure that the amount of resource used for verifying a resource record was bounded, thus protecting the wrapper from problems like circular references and denial of service attacks. If we ignore false positive messages belonging to the class PFP, the average number of security violations that correspond to different domain names is 0.375 per run.

4.9.4 Experiment #2

In Experiment #2, we used a trace of DNS queries received by a name server in an operational environment. Experiment #2 was designed to evaluate the response time of a wrapped name server, the false positive rate of the wrapper, and the computational overhead (i.e., CPU time used) of the wrapper in a practical setting. The results for Experiment #2 show that in practice the average query response time would not be as large as that indicated in Experiment #1 because of local queries and query repetitions.

Data Set

The data set for Experiment #2 consisted of a trace of 1340 DNS queries received by a name server in a "real world" setting. To gather the trace of DNS queries, we modified a name server to log all DNS queries it received and ran it for two days. We also modified the local BIND resolver configuration file to use this name server. In the resolver configuration file, the *search list* was consisted of cs.ucdavis.edu., ucdavis.edu., and ucop.edu. When a BIND resolver is invoked to resolve a *relative* domain name—a domain name that does not have a trailing dot—it appends the domain names in the order specified in the search list and attempt to resolve them until a positive response is received. If none of them results

in a successful resolution, the resolver then generates a query for the relative domain name itself. For example, when the BIND resolver is invoked for domain name host1, it attempts to resolve for host1.cs.ucdavis.edu., host1.ucdavis.edu., host1.ucop.edu., and host1 in that order until a successful resolution is obtained.

Experimental Procedure

The experimental procedure for Experiment #2 was similar to that of Experiment #1 with the following three exceptions. First, we used the trace of 1340 DNS queries we collected over a two-day period rather than the list of domain names corresponding to the 100 web sites. Second, we made our name server an authoritative name server for the domain cs.ucdavis.edu. to take advantage of the domain name locality of the DNS queries. Third, after a name server finished resolving the trace of queries, we recorded the total system CPU times and the total user CPU times used.

Experimental Results

Table 4.3 shows the statistics related to response times recorded by *nslookup* based on 33 runs of this experiment. The mean response time for the wrapped server was 0.12

	Unmodified Name Server				Wrapped Name Server			
# queries	Mean	Min	Max	Std Dev	Mean	Min	Max	Std Dev
200	6.24	3.75	19.78	3.62	8.83	4.27	24.52	4.66
400	11.63	7.44	23.99	4.06	19.56	10.48	34.53	6.37
600	45.59	22.29	147.99	28.31	73.42	40.66	270.41	45.65
800	59.71	35.60	171.83	28.99	94.66	58.53	312.98	49.46
1000	74.15	40.28	263.69	50.93	111.69	70.72	332.60	62.77
1200	96.71	55.36	370.10	75.22	145.10	85.13	396.51	87.50
1340	111.05	70.52	392.48	78.75	165.96	102.38	439.51	91.96

Table 4.3: Cumulative Response Time (in Sec.) for the "Two-day trace" Data Set.

second per query, and that for the unmodified server was 0.08 second per query. Both values are smaller than their counterparts in Experiment #1. We examined the trace segments that correspond to "steep" rises of the graphs marked "wrapper (mean)" and "unmodified (mean)" (e.g., 400th-600th query and 1000th-1340th query), we found that they

could be explained by DNS queries generated by web surfing sessions, which were mostly remote and distinct. Specifically, the trace segment for the 400th-600th query included 43 remote and distinct domain names. The average total response times for those 43 queries for the unmodified server and the wrapped server were 28.29 seconds and 47.54 seconds respectively, which accounted for 83% and 88% of the total response times for that interval respectively.

Table 4.4 shows the CPU times used by the unmodified server and the wrapped server. The figures show that the average CPU times used are a small fraction (8% for

	Unmodified Name Server				Wrapped Name Server			
Type	Mean	Min	Max	Std Dev	Mean	Min	Max	Std Dev
System	4.01	3.43	4.90	0.36	5.32	4.59	5.82	0.33
User	4.35	3.73	4.90	0.26	6.94	6.31	7.90	0.43

Table 4.4: System and User Times Used (in Sec.) for the “Two-day Trace” Data Set.

the unmodified server and 7% for the wrapped server) of the total response time. Thus the increase in response time reported in Table 4.3 was largely due to waiting for response messages sent by remote servers used in the wrapper verification process. The average total CPU time increased from 9.33 seconds to 11.29 seconds (i.e., a 21% increase).

For the two-day trace data set, false positives ranged from 2-10 per run, with the mean being 5.85 and the standard deviation being 1.89. If we only consider false positives in the class GPF, false positives ranged from 0-3 per run, with the mean being 1.18 and the standard deviation being 0.88. Among those false positives in the class GPF, 90% of them were of the “messages sent exceeding the threshold” type, as explained earlier, and the remaining 10% of them were due to timeout during the wrapper verification process.

4.9.5 Experiment #3

The main goal of Experiment #3 is to examine the detection rate of malicious attacks of a wrapped name server (i.e., false negative rate). We investigated the following four types of attacks:

- *Sending incorrect resource records for a remote domain name to the victim*: This is accomplished by using a CNAME resource record in the answer section of a response

message to introduce an arbitrary domain name for which the victim server is not authoritative, and then including incorrect resource records for this remote domain name in the response message. Figure 4.7 shows an example of this type of attacks.

- *Sending incorrect resource records that conflict with the zone data for which the victim is authoritative*: In particular, the attacker uses a CNAME resource record to link to an A resource record for which the victim is authoritative.

- *Sending resource records that correspond to a non-existing domain name that lives in the zone for which the victim is authoritative*: In other words, the authoritative zone does not have this domain name.

- *Sending a response with a query ID that does not match the query ID in the corresponding query sent by the victim*: A variant of this attack is to use a query to trigger the victim name server to send a query to the attacker whom records the query ID used. A second query is then issued to trigger the victim to query the attacker again. Instead of using the query ID used by the victim, the attacker adds one to the query ID used in the first query and uses the result as the query ID in its second response.

The first three types of attacks correspond to sending incorrect DNS data to a name server. These are also called cache-poisoning attacks. The fourth type of attacks corresponds to masquerading attacks. Our wrapper used randomized query ID's for outgoing queries. Thus attackers who do not have access to those queries will have to guess the query ID's used for their forged response messages. As a result, their forged messages will be detected with high probability.

Data Set

In Experiment #3, we modified the data set used in Experiment #2 by inserting two queries that correspond to each of the four types of attacks at random locations in the two-day trace. Moreover, we also inserted four "ordinary" queries at random locations in the trace as controls. These queries correspond to different domain names in the domain for which a malicious name server is authoritative but do not trigger an attack by the malicious name server.

Experimental Procedure

1. Start a malicious name server for a new sub-domain dns.cs.ucdavis.edu. When that malicious name server is asked to resolve for certain domain names that reside in the dns.cs.ucdavis.edu. domain, depending on the domain names queried, it will either return incorrect DNS resource records or send out response messages with an incorrect query ID or a predicted query ID.

2. Start a wrapped name server.

3. Run *nslookup* with the modified trace of DNS queries as input and send the queries sequentially to the wrapped name server.

4. Terminate the wrapped name server.

5. Restart the malicious name server.

6. Start an unmodified name server.

7. Run *nslookup* with the modified trace of DNS queries as input and send the queries sequentially to the unmodified name server.

8. Terminate the unmodified name server.

9. Terminate the malicious name server.

Experimental Results

We ran the experiment five times. In all five runs, all eight attacks (i.e., two from each of the four attack types) were reported correctly by the wrapped name server, and none of the response messages corresponding to the control queries were mis-classified as attacks.

When we applied these four types of attacks to an unmodified name server, the first type of attacks succeeded in planting incorrect DNS data into the cache of the victim server. For the second and the third type, the unmodified name server did not cache the incorrect DNS data for domain names that belong to its authoritative domain. However, the name server did forward the entire response message received, including those incorrect resource records for which the name server was authoritative, to its client. That did not

make much difference for our experiments because the client used was *nslookup*, which did not perform caching. However, if the client was another name server that was not authoritative for those incorrect DNS data, the cache of the client would be corrupted. This situation may occur when the client is a *caching-only server*[31] that uses another name server as a *forwarder*[32]. The "two-query" variant of the fourth type of attacks succeeded for an unmodified name server. It was because the query ID used by an unmodified name server was predictable: the query ID used in successive queries always differed by one.

4.10 Discussion

In our experiments, no false negatives for our wrapper were observed. Specifically, we carried out cache poisoning attacks and a version of spoofing attacks (i.e., query id guessing attacks), and they were all detected by our wrapper. The wrapper prevented the corresponding DNS messages from reaching the protected name server and recorded these messages in its log. On the other hand, those attacks were successful against BIND release 4.9.5. We note that our wrapper would be ineffective if our assumptions were violated. In particular, if an attacker can eavesdrop the query messages sent by our wrapper (hence the randomized query id used), our wrapper may not be effective against spoofing attacks. A strong cryptographic authentication scheme for DNS (e.g., DNSSEC) appears to be necessary to cope these attacks.

For Experiment #2 (i.e., using the two-day trace that contained 1340 queries), there were 5.85 false positives on average. Among these false positives detected by our wrapper, about 80% were caused by misconfigurations of external name servers (e.g., parent and child zones contain inconsistent name server information about the child zone) and the rest were caused by the inability of our wrapper to verify a response message (e.g., timeouts during the wrapper verification process). In other words, the majority of these false positives are in the class PFP (as defined in Section 4.9). In many cases, false positives in PFP and malicious attacks are virtually indistinguishable without performing an "out of band" investigation. Thus it is not clear how we can significantly reduce the number of false positives.

[31] A caching-only server is a name server that is not authoritative for any domain.

[32] A forwarder is a name server to which other name servers forward their recursive queries. A forwarder is useful for building a large cache for remote DNS data, especially when communication between local machines and remote machines is slow or restricted.

There is a moderate increase in response time when the wrapper is used. Specifically, for Experiment #1 (which involves 100 remote and distinct domain names) the mean response time per query increased from 0.4791 second to 0.8546 second. For Experiment #2, the mean response time per query increased from 0.08 second to 0.12 second. The percentage increases in the response time seem to be moderately large; however, the impact on the users is insignificant because of the small absolute values. As for CPU overhead, we observed a 20% increase in CPU load due to the wrapper in Experiment #2. Because name servers are usually not CPU-bound, the wrapper should be applicable in most environments.

4.11 Summary and Future Work

In this chapter, we examined security vulnerabilities of DNS, evaluated existing approaches for protecting DNS, presented a detection-response approach for protecting DNS, developed formal specifications for DNS clients, DNS servers, and our DNS wrapper, and performed experiments to evaluate the performance for our wrapper prototype: the number of false negatives, the number of false positives, the changes in response time, and the computational overhead associated with deploying the DNS wrapper.

Our approach consists of the following steps. First, we define our security goal—name servers only use DNS data that are consistent with the corresponding authoritative data. Second, we develop a DNS model and write formal specifications for DNS clients and DNS servers. Third, we design our DNS wrapper with the objective that the composition of the specification for a protected name server and that for the wrapper satisfy our security goal for DNS. If the DNS wrapper receives a DNS message that may cause violations of our security goal, the wrapper drops the message instead of forwarding it to the protected name server. Fourth, we use the formal specification for our wrapper to guide our implementation of a wrapper prototype.

Compared to the prior work for protecting DNS, our DNS wrapper has the following advantages:

- Provides assurance by employing formal methods.
- Effective against cache poisoning attacks and certain spoofing attacks (i.e., query id guessing).

- Compatible with existing DNS implementations.
- Does not require changes for the DNS protocol.
- Incurs reasonable performance overhead.
- Can be deployed locally; does not depend on changes to other remote DNS components.

In November 1998, a company called Men & Mice surveyed the status of domain name services on the Internet [42]. Among 4184 randomly picked com zones, they found that 1344 of them (i.e., 32.1%) were vulnerable to cache poisoning attacks. In other words, the name servers for those zones could be compromised and gave out incorrect information about other domains, including its delegated domains. We note that the effectiveness of our DNS wrapper is not affected by attacks against external name servers as long as our assumptions are met.

There are several directions for future research.

- To further raise the assurance level of our wrapper, we may perform a complete formal verification from specification to implementation. The VDM specifications developed in this chapter could be the basis for conducting the formal verification.
- Results from Experiment #2 show a 0.437% false positive rate for the DNS wrapper. Because the majority of these false positives are caused by misconfigurations of external name servers, a non-trivial modification for the DNS wrapper may be needed to significantly reduce the false positive rate.
- We have not discussed protecting DNS resolvers. If the communication link between a resolver and its trusted local name server is secure, and the name server is protected by our DNS wrapper, the DNS data received by the resolver is "safe" because a wrapped name server only uses DNS data that are consistent with the corresponding authoritative answers. Future research may be conducted for protecting DNS resolvers when the resolver-server communication link is insecure. A possibility is to adapt our DNS wrapper to protect resolvers.
- We may apply our approach to protect other network services and privileged processes.

Chapter 5

Conclusions and Future Work

> "Goodbye Jean-Luc, I'm gonna miss you. You had such potential. But then again, all good things must come to an end."
>
> —Q (Star Trek: The Next Generation)

5.1 Conclusions

This section summarizes the results and the contributions of this dissertation. Section 5.1.1 presents our intrusion tolerance approach. Our approach, a significant extension of intrusion detection, includes detection of violations of security policies, system diagnosis to identify misbehaving components, and automated response to prevent an attack from propagating and to restore the operational status of the system. Formalism is an integral part of our approach. To raise the assurance level of detection-based solutions, our approach includes modeling of system components, characterizing system components using formal specifications, and proving properties of the solutions. We have applied our intrusion tolerance approach to three sub-problems in securing a network infrastructure: Section 5.1.2 summarizes our efficient message authentication scheme for link state routing. Section 5.1.3 summarizes our work in detecting and responding to routers that maliciously drop packets and misroute packets. Section 5.1.4 summarizes our wrapper-based solution for protecting domain name systems.

5.1.1 Intrusion Tolerance: A New Approach

This dissertation presented an intrusion tolerance approach for protecting network infrastructures. Our approach, which extends prior work on intrusion detection and fault tolerance, includes the following:

- Cooperation among network components to detect attacks that are beyond the capability of any single component (e.g., in the flow analysis protocol, routers that are neighbors of a tested router cooperate to detect whether the tested router drop packets).
- System diagnosis to identify misbehaving network components (e.g., in the Optimistic Link State Verification scheme, routers exchange information and trace the propagation of forged routing updates to identify misbehaving routers).
- Automated response to prevent misbehaving components from affecting other components (e.g., a DNS wrapper that drops messages that may compromise a name server) or to restore the operational status of a system by system reconfiguration (e.g., a routing protocol that logically disconnects misbehaving routers).

In current practice, when potential attacks are detected, intrusion detection systems will send a warning to a security administrator, who will be responsible for investigating and resolving the problems. We envision that human intervention at this level will not be feasible for much longer because of an increasing number of attacks, increasing use of automated attacks (hence potentially rapid propagation of attacks), and increasing costs attendant to slow human response. For instance, having a network (infrastructure) unavailable—even for a short time—will incur high costs as more and more users rely on computer networks to perform time critical activities. Our approach includes, in addition to detection, system diagnosis to identify misbehaving network components and automated response such as system reconfiguration to avoid using components that are diagnosed as misbehaving.

Our approach is novel (among intrusion detection work) in using system diagnosis to identify misbehaving components and reconfiguration for system recovery. Our detection-diagnosis-reconfiguration approach is useful when one of the following items hold:

- *Prevention measures are too expensive or too restrictive to use:* Implementing security measures to prevent attacks could be expensive. By using a detection-recovery

approach, we may only need to invoke expensive measures when the system is under attack. We note that this approach may not be applicable in some circumstances. For example, when there is leakage of confidential information, the damage may be irrecoverable. Chapter 2 illustrated when detection-based solutions may be used as an alternative to prevention, namely in the context of efficient message authentication for link state routing.

- *Retrofitting is used to improve the security of existing systems:* To live with existing network protocols and legacy systems, changes to them should be kept at a minimum when we improve their security. Chapter 3 illustrated how our approach can be used in existing link state routing protocols to detect and to respond to routers that incorrectly drop or misroute transit packets.

Applying our approach to protect network infrastructures should result in solutions that are more attack resistant (or survivable) than existing intrusion detection approaches. Through the use of cooperative detection, decentralized diagnosis, and autonomous response, our solutions are less susceptible to a single point of failure and more capable of achieving faster response. For example, if a router is compromised, it may masquerade as a trusted router and send out erroneous routing control messages. In our solutions, after cooperatively detecting an attack and performing diagnosis to identify misbehaving routers, the neighbors of a compromised router respond by terminating the neighbor relationships with the offensive router, thus logically disconnecting it from the network. As a result, the damage caused by the attack would be confined to those machines that are directly connected to that compromised router. The rest of the network can still function, although perhaps at a degraded level. On the other hand, existing intrusion detection systems often assume that their components themselves are reliable. For example, centralized security managers are used in most existing intrusion detection systems to aggregate and to analyze data sent by individual data sources. If a data source or a centralized security manager is compromised, an attacker could hide the attack by having that compromised component submit erroneous reports.

Formalism is an integral part of our approach, which includes modeling of system components, capturing the functionality of system components by formal specifications, and proving properties of our solutions. Previously, formal methods have not been used in connection with intrusion detection. A common concern for intrusion detection systems is

that it is difficult to assess the benefits of deploying those systems. Currently, a testing-based methodology (e.g., [54, 55]) is used to evaluate an intrusion detection system, namely by subjecting it to test data that contain attacks and to test data that are normal/attack-free. Results like the detection rate for real attacks and the false alarm rate are used to assess the effectiveness of an intrusion detection system. The usefulness of the testing-based methodology depends on the coverage of the test data and the sensitivity of the test results with respect to changes in the operating environments. However, it is hard to obtain test data from a real environment that are known to be free of attacks (for control purposes) and test data that contain attacks representing most attack categories (for coverage). Attacks experienced by a system quite often change with time as variations of known attacks surface or new attacks emerge. Thus it is difficult to develop a set of test data that is comprehensive and that has strong prediction power with respect to changes in the operating environments. At the policy enforcement level, it is often difficult, if not impossible, to construct the set of enforced security policies given a set of attack signatures (as in misuse detection) or low level system call traces (as in specification-based intrusion detection [31, 32, 61]). Moreover, it is difficult to determine if an existing security policy is enforced based on attack signatures and low level system call traces. The testing-based methodology is a very useful means to evaluate intrusion detection systems. However, a more formal methodology is needed to complement the testing-based methodology to facilitate reasoning with respect to high level security policies and thus to raise the assurance level of detection-based solutions. Our work is an initial step towards such a formal methodology.

5.1.2 On Efficient Message Authentication for Link State Routing

Securing a routing infrastructure necessitates assuring the authenticity of routing control messages communicated among routers. Routers construct their routing tables based on those control messages (which describe the current states of the corresponding routers) to cooperatively forward packets from their sources to their destinations. If forged control messages are used by routers, the routing infrastructure might be configured so it is unable to perform its function correctly and efficiently. For example, erroneous control messages might configure routers so that packets will take longer to or never reach their destinations. As pointed out in [47] and in Section 2.2, existing message authentication schemes are either too expensive computationally or too restrictive to be used for protecting

routing control messages in link state routing.

We presented a novel detection-based message authentication scheme for link state routing. Our scheme is efficient and does not have those restrictions. Our scheme is based on an observation that most of the time routing infrastructures are not compromised. Instead of expending considerable computational resource in securing routing control packets on a continuous basis, our detection-based message authentication scheme incurs little resource cost during normal operations, but does use more resources when routing infrastructures are under attack. Our scheme is up to two orders of magnitude faster than an MD5/RSA digital signature scheme during normal operations—by far the more common situation. The technique used in our message authentication scheme may be applicable to applications that need lightweight secure multicasting (c.f. Section 5.2.1). For example, content providers can use our techniques in the secure multicast of news articles, stock quotes, and advertisements.

5.1.3 On Protecting Routing Infrastructures from Denial of Service

Even though end users can employ encryption and message authentication to protect the privacy and the integrity of messages sent over an insecure network, no prior research has been conducted to protect these messages from denial of service attacks at the routing infrastructure level. Denial of service can be caused by misbehaving—faulty, misconfigured, or compromised—routers that misroute or drop packets.

To fill this gap, we developed failure models for routers according to the types of packet information that might be used to select potential victim packets (e.g., packets with certain source addresses are mishandled) and to the amount of packets that are mishandled (e.g., ten percent of the packets destined for foo.edu are mishandled). Based on these failure models, we developed techniques and protocols to detect, identify, and respond to misbehaving routers. Our protocols are distributed and do not assume a centralized manager that collects and analyzes the test results. We formally proved important properties of our protocols under a set of assumptions. Specifically, our protocols are *sound* (i.e., a good router never incorrectly claims another router as misbehaving), *complete* (i.e., if there are misbehaving routers in a network, one or more of them can be identified), and *responsive* (i.e., all routers that misbehave infinitely often will eventually be removed).

In the field of intrusion detection, we tend to assume the worst case scenarios

(e.g., attackers have complete control over a compromised system). While solutions that are based on the "omniscient" and "omnipotent" attacker assumption are more secure than those that based on a more restricted assumption, the worst case assumption usually leads to a solution that is significantly more complicated to design and to analyze, more restrictive to use, and more expensive to deploy. Our study of different failure models is inspired by the use of fault models of different "strength" in fault tolerance work. It may be prudent to study different characteristics (or strengths) of failure modes in security because some failure modes may be more difficult for an attacker to achieve than others. For example, use of a proprietary operating system for a router may make attacks that require modifications of router software more difficult than those that require modifications of routing tables. In Chapter 3, we presented two techniques to cope with failure models of different strengths, namely distributed probing and flow analysis. Flow analysis assumes a more malicious failure model (i.e., intermittent and content-aware failure) than distributed probing does. However, the flow analysis technique is more expensive to use than the distributed probing technique because of the need to monitor every transit packet for every router. Moreover, our protocol that is based on the flow analysis technique is also more complicated than those based on the distributed probing technique.

5.1.4 On Protecting Domain Name Systems

In addition to securing routing infrastructures, this dissertation also presented a detection-response approach for protecting another core component of network infrastructures, namely domain name systems (DNS). Unlike some prior work that requires changes in remote name servers or extensions for the DNS protocol, our solution can be deployed independently with individual name servers and does not require any changes in the DNS protocol.

Our approach is driven by formal specifications. We developed functional specifications (in VDM) for DNS clients and DNS servers. Basically, DNS components are modeled as an object that maintains a mapping for DNS data. The mapping may be changed only through communicating with other DNS components (i.e., sending DNS requests and receiving DNS responses) or by timeouts for DNS data. We specified our security goal for DNS: Name servers only use DNS data that are consistent with those originating in the corresponding authoritative name servers. Then we characterized a DNS wrapper, which

enforces this security goal, using formal specifications. Our DNS wrapper filters out DNS messages that may cause violations of our security goal instead of forwarding them to a protected name server. Based on the DNS wrapper specification, we implemented a DNS wrapper prototype and evaluated its performance.

From our experiments, we found that the increase in response time and the CPU overhead due to the wrapper were reasonable. For example, in the second set of experiments, the average per query response time increased from 0.08 second to 0.12 second, and the average per run CPU time increased from 9.33 seconds to 11.29 seconds. In one of our experiments, we found that the average false positive rate was 0.437%. About 80% of these false positives were caused by misconfigurations of external name servers. Our DNS wrapper is effective against cache poisoning attacks and certain spoofing attacks. This was confirmed by our experimental results that showed a zero false negative rate with respect to our test data.

To sum up, this dissertation makes four main contributions. First, we presented an intrusion tolerance approach for protecting network infrastructures. Second, we presented an efficient detection-based message authentication scheme for link state routing. Third, we presented techniques and protocols for detecting, identifying, and responding to routers that misroute packets and drop packets incorrectly. Fourth, we presented a wrapper-based solution for protecting domain name systems.

5.2 Future Work

5.2.1 On Optimistic Link State Verification

The optimistic link state verification (OLSV) scheme introduced in Chapter 2 is based on the observation that network infrastructures operate normally much more often than under attack. Thus one can afford to employ more expensive diagnosis and reconfiguration procedures to recover from an attack as they are used infrequently. To understand the operating limit of OLSV, we may perform simulations for running OLSV or evaluate OLSV analytically using a more detailed mathematical model that includes, among other things, the frequency of successful router attacks and the frequency of link state changes.

Even though OLSV does not require ϵ (the maximum clock skew between any

two good routers) to be smaller than a certain threshold, we note that there is a tradeoff between the tightness of clock synchronization and the time to recover: The elapsed time between releasing a signed link state update (LSU) and releasing the corresponding hash-chained key (HCK) is greater than 2ϵ time units. Thus the larger the ϵ, the longer routers take to detect a forged LSU. A direction for future research is to reduce the time to recover for OLSV when routers are loosely synchronized. If we cannot use a secure network time protocol to reduce clock skew among routers, a plausible solution is to extend OLSV in the following way: Because LSUs are broadcasted to every router, a router can detect a forged LSU that claim to originate from itself. In such a case, the router can sign the corresponding hash-chained key using its public key and broadcast the signed key to other routers to initiate the recovery procedure sooner.

We may apply our efficient message authentication techniques to some lightweight secure multicast applications. For example, a content provider may need to push news articles and stock quotes to many subscribers. Using conventional digital signatures to sign and to verify the integrity of all those messages may be too costly, especially on the subscriber side because a subscriber may not have a fast machine and other applications may be running on that machine when the message verification is being performed. Also, using a symmetric-key based message authentication scheme may be inefficient because the content provider needs to sign and to send a dedicated message to each subscriber. Our techniques enable an efficient solution for this problem. Specifically, a subscriber machine may first synchronize its clock with the content provider's clock or engage in a protocol to find out the clock skew between them. We assume that the public key of the content provider is signed by some certificate authorities trusted by the potential subscribers. The content provider generates a hash chain, constructs and signs a key-chain anchor (KCA) message using its private key, and distributes the signed KCA to its subscribers. Then the content provider can use the HCKs to generate the message authentication codes for its multicasted messages. If a user subscribes to the channel after some HCKs are used, the content provider may sign the most recently disclosed hash-chained key using its private key and send the signed key, which serves a similar function as a signed KCA does, to the subscriber. If the clock of the content provider and those of the subscribers are tightly synchronized, or the subscribers can tolerate a certain time delay between the dissemination and the use of a message, one may design a message authentication scheme that does not use "optimistic verification": After the subscriber machine receives a message signed by

an HCK, it will wait until the HCK arrives. After the HCK and the message are verified, the machine will deliver the message to the subscriber.

5.2.2 On Protecting Routing Infrastructures from Denial of Service

In Chapter 3, we made a simplifying assumption that a good router does not drop packets. However, when network congestion occurs, routers may be forced to drop packets. We propose to handle this by incorporating a threshold on the amount of transit traffic a good router can drop to our protocols. We note that using a threshold scheme is not perfect. For example, if a bad router only drops few packets, a threshold scheme may miss the attack. On the other hand, it is probably useful for coping with large scale denial of service attacks. For distributed probing, one can set a threshold so that a router is diagnosed as good if it passes a certain percentage of tests. For flow analysis, one can adapt our conservation of transit traffic test so that a router is diagnosed as bad if the difference between its incoming and its outgoing transit traffic is larger than a certain threshold. Setting the threshold depends on many factors such as network load and traffic patterns, network topology, and the capacities of routers and communication links. Determining the value of a proper threshold is a future research issue.

Another research direction one may pursue is to expand the scope of our work (in Chapter 3) to include other failure models. Suppose a misbehaving router may modify the payload of transit packets. Our distributed probing protocols can be adapted to cope with packet payload modification attacks by checking the integrity of test packets after their return to the tester. Our flow analysis protocol can cope with packet payload modification if there is a change in packet length. However, extending our flow analysis protocol to efficiently cope with attacks like flipping random bits in transit packets appears to be difficult. Computing message authentication codes[1] (MAC) for transit packets seems to be useful to detect bit flipping attacks. However, the overheads for generating, transmitting, and verifying these MAC for every transit packet are quite high. And finding a means to combine the MAC for successive transit packets to reduce the overheads is a challenge. This is because a router may need to intermix transit packets sent by different neighboring routers and a router may reorder or drop packets occasionally. It may be worthwhile to revisit the distributed probing technique for coping with more malicious failure models.

[1] The argument here holds for using cryptographic checksums (e.g., digital signatures) in general.

5.2.3 On Protecting Domain Name Systems

In Chapter 4, we used VDM to specify DNS clients, DNS name servers, and our DNS wrapper. To further raise the assurance level of our solutions, one may perform a complete formal verification to ensure the transformations from specification to implementation was done correctly. VDM not only is well-suited for formal specification, but also includes a proof theory that can be used to conduct rigorous inference from specification to code concerning the properties of a specified system.

Another direction to pursue is to reduce the false positive rate for the DNS wrapper. The average number of false positives for the second set of experiments (which used a two-day trace for one user with 1340 queries) was 5.85. Even though the false positive rate does not appear to be very high, it might be a problem for a security administrator who manages a large site. Our experimental results indicated that about 20% of the false positives were caused by timeouts and a threshold we imposed to prevent the DNS wrapper from spending too much resources on verifying one packet. One may try to fine tune these parameters to reduce the false positive rate. A more challenging problem is to cope with the remaining 80% of the false positives that were caused by misconfigurations of external name servers. Some of them may be virtually indistinguishable from an attack. A possible approach is to build a rule base that describes the common categories of misconfigurations and to change the DNS wrapper to remove the offending resource records quietly (instead of generating a warning) when DNS messages in those categories are received.

Bibliography

[1] P. Albitz, and C. Liu, "DNS and BIND." O'Reilly and Associates, Inc., 1992.

[2] D. Andrews, and D. Ince, "Practical Formal Methods with VDM." McGraw-Hill, 1991.

[3] M. Barborak, M. Malek, and A. Dahbura, "The Consensus Problem in Fault-Tolerant Computing." *ACM Computing Surveys*, Vol.25, No.2, June 1993, pp.171-220.

[4] S.M. Bellovin, "Security Problems in the TCP/IP Protocol Suite." *Computer Communications Review*, Vol.19, No.2, April 1989, pp.32-48.

[5] S. Bellovin, "Using the Domain Name System for System Break-ins." *Proc. of the* 5th *UNIX Security Symposium,* June 5-7, 1995, pp.199-208.

[6] K.A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R.A. Olsson, "Detecting Disruptive Routers: A Distributed Network Monitoring Approach." *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, Oakland, California, May 3-6, 1998, pp.115-124.

[7] K.A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R.A. Olsson, "Detecting Disruptive Routers: A Distributed Network Monitoring Approach." *IEEE Network*, Vol.12, No.5, September/October 1998, pp.50-60.

[8] CERT Coordination Center, "IP Spoofing Attacks and Hijacked Terminal Connections." CERT Advisory CA-95:01, January 23, 1995.

[9] CERT Coordination Center, "*smurf* IP Denial-of-Service Attacks." CERT Advisory CA-98:01, January 5, 1998.

[10] CERT Coordination Center, "Multiple Vulnerabilities in BIND." CERT Advisory CA-98:05, April 8, 1998.

[11] CERT Coordination Center, "Melissa Macro Virus." CERT Advisory CA-99:04, March 27, 1999.

[12] W.R. Cheswick, and S.M. Bellovin, "Firewalls and Internet Security: Repelling the Wily Hacker." Addison-Wesley, 1994.

[13] B. Cheswick, and S. Bellovin, "A DNS Filter and Switch for Packet-filtering Gateways." *Proc. of the* 6th *UNIX Security Symposium,* July 22-25, 1996, pp.15-19.

[14] S. Cheung, "An Efficient Message Authentication Scheme for Link State Routing." *Proceedings of the* 13th *Annual Computer Security Applications Conference*, San Diego, California, December 8-12, 1997, pp.90-98.

[15] S. Cheung, and K.N. Levitt, "Protecting Routing Infrastructures from Denial of Service Using Cooperative Intrusion Detection." *Proceedings of the New Security Paradigms Workshop*, Cumbria, UK, September 23-26, 1997, pp.94-106.

[16] D. Comer, "Internetworking with TCP/IP." Vol.1, Prentice Hall, 1991.

[17] The Fourth Workshop on Computer Misuse and Anomaly Detection (CMAD IV), Monterey, California, November 12-14, 1996.

[18] D. Denning, "An Intrusion-Detection Model." *IEEE Transactions on Software Engineering*, Vol.SE-13, No.2, February 1987, pp.222-232.

[19] D. Eastlake, 3rd, and C. Kaufman, "Domain Name System Security Extensions." RFC 2065, January 1997.

[20] G.G. Finn, "Reducing the Vulnerability of Dynamic Computer Networks." ISI Research Report RR-88-201, University of Southern California, June 1988.

[21] S. Forrest, A.S. Perelson, L. Allen, and R. Cherukuri, "Self-Nonself Discrimination in a Computer." *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, Oakland, California, May 16-18, 1994, pp.202-212.

[22] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A Sense of Self for Unix Processes." *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, California, May 6-8, 1996, pp.120-128.

[23] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS Software with Generic Software Wrappers." To appear in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 5-7, 1999.

[24] J.M. Galvin, "Public Key Distribution with Secure DNS." *Proc. of the* 6th *UNIX Security Symposium,* July 22-25, 1996, pp.161-170.

[25] E. Gavron, "A Security Problem and Proposed Correction with Widely Deployed DNS Software." RFC 1535, October 1993.

[26] R. Hauser, T. Przygienda, and G. Tsudik, "Reducing the Cost of Security in Link-State Routing." *Proceedings of the Symposium on Network and Distributed System Security (SNDSS '97)*, San Diego, California, February 10-11, 1997, pp.93-99. A journal version will appear in *Computer Networks and ISDN Systems.*

[27] K. Ilgun, R.A. Kemmerer, and P.A. Porras, "State Transition Analysis: A Rule-Based Intrusion Detection Approach." *IEEE Transactions on Software Engineering*, Vol.21, No.3, March 1995, pp.181-199.

[28] Joint Technical Committee ISO/IEC JTC 1 *Information Technology*, "Intermediate System to Intermediate System Intra-domain Routeing Information Exchange Protocol for Use in Conjunction with the Connectionless-mode Network Service (ISO 8473)." ISO/IEC 10589, April 1992.

[29] C.B. Jones, "Systematic Software Development using VDM." Prentice-Hall, 1990.

[30] S. Kent, and R. Atkinson, "Security Architecture for the Internet Protocol." RFC 2401, November 1998.

[31] C. Ko, G. Fink, and K. Levitt, "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring." *Proc.* 10th *Annual Computer Security Applications Conference*, Orlando, Florida, Dec. 1994, pp.134-44.

[32] C. Ko, M. Ruschitzka, and K. Levitt, "Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-Based Approach." *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, California, May 5-7, 1997, pp.134-144.

[33] H. Krawczyk, M. Bellare, and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication.” RFC 2104, February 1997.

[34] B. Kumar, and J. Crowcroft, “Integrating Security in Inter-domain Routing Protocols.” *Computer Communication Review*, Oct. 1993, Vol.23, No.5, pp.36-51.

[35] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System.” *Communications of the ACM*, Vol.21, No.7, July 1978, pp.558-565.

[36] L. Lamport, “Password Authentication with Insecure Communication.” *Communications of the ACM*, Vol.24, No.11, November 1981, pp.770-772.

[37] P.A. Lee, and T. Anderson, “Fault Tolerance: Principles and Practice.” Springer-Verlag, Second Edition, 1990.

[38] T.F. Lunt, “A Survey of Intrusion Detection Techniques.” *Computer and Security*, June 1993, Vol.12, No.4, pp.405-418.

[39] G. Malkin, “RIP Version 2 — Carrying Additional Information.” RFC 1723, November 1994.

[40] J. McQuillan, I. Richer, and E. Rosen, “The New Routing Algorithm for the ARPANET.” *IEEE Transactions on Communications*, Vol.COM-28, No.5, May 1980, pp.711-719.

[41] C. Meadows, “The Need for a Failure Model for Security.” *Proceedings of the* 4th *Conference on Dependable Computing for Critical Applications*, San Diego, California, January 4-6, 1994, pp.383-385.

[42] Men and Mice, “Domain Health Survey.” http://www.menandmice.com/dnsplace/healthsurvey.html, November 1998.

[43] P. Mockapetris, “Domain Names – Concepts and Facilities.” RFC 1034, November 1987.

[44] P. Mockapetris, “Domain Names – Implementation and Specification.” RFC 1035, November 1987.

[45] J. Moy, “OSPF Version 2.” RFC 2178, July 1997.

[46] B. Mukherjee, L.T. Heberlein, and K.N. Levitt, "Network Intrusion Detection." *IEEE Network*, May-June 1994, Vol.8, No.3, pp.26-41.

[47] S.L. Murphy, and M.R. Badger, "Digital Signature Protection of the OSPF Routing Protocol." *Proceedings of the Symposium on Network and Distributed System Security (SNDSS '96)*, San Diego, California, February 22-23, 1996, pp.93-102.

[48] NIST (National Institute of Standards and Technology). "Secure Hash Standard." NIST FIPS Pub. 180, May 1993.

[49] PC Magazine: Top 100 Web Sites, http://www.zdnet.com/pcmag/special/web100, 1998.

[50] R. Perlman, "Network Layer Protocols with Byzantine Robustness." Ph.D. Dissertation, Massachusetts Institute of Technology, August 1988.

[51] R. Perlman, "Interconnections: Bridges and Routers." Addison-Wesley, 1992.

[52] F.P. Preparata, G. Metze, and R.T. Chien, "On the Connection Assignment Problem of Diagnosable Systems." *IEEE Transactions on Electronic Computers*, Vol.EC-16, No.6, December 1967, pp.848-854.

[53] T. Ptacek, and T. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection." Technical reports, Secure Networks, Inc., January 1998.

[54] N. Puketza, B. Mukherjee, R.A. Olsson, and K. Zhang, "Testing Intrusion Detection Systems: Design Methodologies and Results from an Early Prototype." *Proceedings of the* 17th *National Computer Security Conference*, October 1994, pp. 1-10.

[55] N.F. Puketza, K. Zhang, M. Chung, B. Mukherjee, R.A. Olsson, "A Methodology for Testing Intrusion Detection Systems." *IEEE Transactions on Software Engineering*, Vol.22, No.10, October 1996, pp.719-729.

[56] R. Rivest, "The MD5 Message-Digest Algorithm." RFC 1321, April 1992.

[57] R. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." *Communications of the ACM*, Vol.21, No.2, February 1978, pp.120-126.

[58] E. Rosen, "Vulnerabilities of Network Control Protocols: An Example." RFC 789, September 1981.

[59] A. Salamon, "Name Server Software Summary."
http:://www.dns.net/dnsrd/docs/exotic.html, January 1997.

[60] C.L. Schuba, and E.H. Spafford, "Addressing Weaknesses in the Domain Name System Protocol." Technical Report, Department of Computer Sciences, Purdue University, 1994.

[61] R. Sekar, Y. Cai, and M. Segal, "A Specification-Based Approach for Building Survivable Systems." *Proc.* 21st *National Information Systems Security Conference*, Arlington, Virginia, October 6-9, 1998.

[62] K.E. Sirois, and S.T. Kent, "Securing the Nimrod Routing Architecture." *Proceedings of the Symposium on Network and Distributed System Security (SNDSS '97)*, San Diego, California, February 10-11, 1997, pp.74-84.

[63] B.R. Smith, and J.J. Garcia-Luna-Aceves, "Securing the Border Gateway Routing Protocol." *Proceedings of Global Internet 1996*, London, England, November 20-21, 1996.

[64] B.R. Smith, S. Murthy, and J.J. Garcia-Luna-Aceves, "Securing Distance-Vector Routing Protocols." *Proceedings of the Symposium on Network and Distributed System Security (SNDSS '97)*, San Diego, California, February 10-11, 1997, pp.85-92.

[65] E.H. Spafford, "Crisis and Aftermath." *Communications of the ACM*, Vol.32, No.6, June 1989, pp.678-687.

[66] M. Steenstrup, "Routing in Communications Networks." Prentice Hall, 1995.

[67] Trusted Information Systems, "TIS/DNSSEC."
http:://www.tis.com/docs/research/network/dns.html, 1997.

[68] G. Tsudik, "Message Authentication with One-Way Hash Functions." *Proceedings of IEEE INFOCOM '92*, Florence, Italy, May 4-8, 1992, pp.2055-2059.

[69] W. Venema, "TCP Wrapper: Network Monitoring, Access Control, and Booby Traps." *Proc. of the* 3rd *UNIX Security Symposium,* September 1992, pp.85-92.

[70] P. Vixie, "Name Server Operations Guide for BIND (Release 4.9.3)."

[71] P. Vixie, "DNS and BIND Security Issues." *Proc. of the* 5th *UNIX Security Symposium,* June 5-7, 1995, pp.209-216.

[72] S.F. Wu, F. Wang, B.M. Vetter, W. Rance Cleaveland II, Y.F. Jou, F. Gong, and C. Sargor, "Intrusion Detection for Link-State Routing Protocols." Presentation in the *1997 IEEE Symposium on Security and Privacy*, Oakland, California, May 4-7, 1997. Further information is available at http://shang.csc.ncsu.edu.

Appendix A

BIND 4.9.5 Resource Record Filtering Algorithm

```
We extracted the following resource record filtering algorithm
from ns_resp.c of BIND release 4.9.5.

Notation:
  name(r) --- name associated with resource record r
  type(r) --- type associated with resource record r
  rdata(r) --- data associated with resource record r
  For example:
    If resource record r is "h1.cs.ucdavis.edu  CNAME  h2.cs.ucdavis.edu"
    Then name(r) = h1.cs.ucdavis.edu
         type(r) = CNAME
         rdata(r) = h2.cs.ucdavis.edu

Algorithm:
  cname = lastwascname = 0
  aname = query name in the question section
  for each resource record r in the answer section
     if name(r) != aname
        drop r
        next resource record
     if type(r)==CNAME and CNAME type is not what we want
        aname = rdata(r)
        lastwascname = cname = 1
     else
        lastwascname = 0
     if r has the auth answer flag set and
        name(r) == name in the question section
```

```
      r is an authoritative answer
   else
      r is an answer

for each resource record r in the authority section
   if lastwascname
      drop r
      goto end
   if type(r)==NS
      if (aname does not belong to the domain denoted by name(r)) OR
         (cname==0 AND name(r) does not belong to the domain of the
          responding server)
         drop r
         next resource record
   if the query is for system "priming" (i.e., initialize the cache with
      a list of root server addresses)
      r is an answer
   else
      r is an additional resource record.

for each resource record r in the additional section
   if the query is for system "priming"
      r is an answer
   else
      r is an additional resource record.

end:
```

Appendix B

Additional Specifications for Our Wrapper

This appendix contains specifications for our DNS wrapper that are not detailed in Chapter 4.

$KnownAuthServer(d : DName,\ c : Class,\ m : DbMap)\ ss : Server-\mathsf{set}$

/* Returns a set of authoritative name servers (ss) for domain d and class c with both the NS and the A resource records in m */

/* Used by *AuthVerified* */

$\mathsf{post} : ss = \{s \mid \exists arr,\ nsrr \in \bigcup \mathsf{rng}\ m \cdot$

$dname(nsrr) = d \wedge dname(arr) = rdata(nsrr) \wedge dname(arr) = Sname(s) \wedge$

$type(nsrr) = \mathsf{NS} \wedge type(arr) = \mathsf{A} \wedge class(nsrr) = class(arr) = c\}$

/* Sname(s) gives the domain name corr. to server s */

$MatchCount(a : DName,\ b : DName)\ count : Integer$

/* Used by *NearestZoneKnown* */

$\mathsf{post} : \mathsf{if}\ suffix(a, b)\ /*\ if\ a\ is\ a\ suffix\ of\ b\ */$

$\mathsf{then}\ count = length(a);\ /*\ num\ of\ labels\ in\ a\ */$

$\mathsf{else}\ count = 0;$

$NearestZoneKnown(p : Process,\ rr : RR)\ nz : Zone$

/* For a resource record rr, return the nearest zone of which we know an auth name server */

/* Used by *AuthVerified* */

$\mathsf{post} : \mathsf{let}\ zs = \{z \mid \exists rr_1, rr_2 \in View(p) \cdot$

$(dname(rr_1), type(rr_1), class(rr_1)) = (z, \mathsf{NS}, class(rr)) \wedge$

$(dname(rr_2), type(rr_2), class(rr_2)) = (rdata(rr_1), \mathsf{A}, class(rr))\}\ \mathsf{in}$

$nz \in \{z \in zs \mid \forall z_1 \in zs \cdot MatchCount(z, dname(rr)) \geq$

$MatchCount(z_1, dname(rr))\}$

$ConstructQ_w(d : DName,\ t : RRType,\ c : RRClass)\ q : Query$

/* returns query q with the input parameters that has a random query id and the rd bit unset to request iterative resolution */

/* Used by CheckAuthServer */

pre : $(\mathsf{rng}\ transTable) \neq QID$ /* unused query id available */

post : $Q(q) = (d, t, c) \wedge rd(q) = \mathsf{false} \wedge id(q) = randomId(\{i \mid i \notin \mathsf{rng}\ transTable\})$

$Get_w(q : Query,\ to : Server)\ m : Resp$

/* sends query q to server to. returns null when there is no response (i.e., timeout); otherwise, returns the response that corresponds to the query q.*/

/* Used by CheckAuthServer */

pre : true

post : $(m = \mathsf{null} \vee (m \neq \mathsf{null} \wedge Q(q) = Q(m)))$

$IsAuthDomain(rr : RR,\ d : Domain)\ auth : Boolean$

/* returns true if d is the authoritative domain for rr; returns false otherwise.*/

/* Used by CheckAuthServer */

pre : true

post : let $zs = \{z \mid \exists rr_2 \in \bigcup(\mathsf{rng}\ Db) \cdot (dname(rr_2), type(rr_2), class(rr_2)) = (z, \mathsf{NS}, class(rr))\}$ in

$auth = (DtoZ(d) \in \{z \in zs \mid \forall z_1 \in zs \cdot MatchCount(z, dname(rr)) \geq MatchCount(z_1, dname(rr))\})$

$\mathsf{GETNS-A} \sqsubseteq$

/* The macro GETNS-A, used by *CheckAuthServer*, retrieves the NS resource records and the A resource records for a certain domain, *newcurdom*. m is a DNS response containing information for *newcurdom* */

/* Used by CheckAuthServer */

nss = $\{nrr \mid nrr \in Auth(m) \wedge$

$(dname(nrr), type(nrr), class(nrr)) = (newcurdom, \mathsf{NS}, class(nrr))\}$;

foreach $nrr \in nss\cdot$ /* collect NS rr for newcurdom */

if $(dname(nrr), type(nrr), class(nrr)) \in dom(newrs)$

then newrs(dname(nrr),type(nrr),class(nrr)) =

newrs(dname(nrr),type(nrr),class(nrr)) $\cup \{nrr\}$;

else newrs(dname(nrr),type(nrr),class(nrr)) = $\{nrr\}$;

foreach $arr \in Add(m)\cdot$ /* collect glue A rr for NS in nss */

if $type(arr) = A \wedge \exists nrr \in nss \cdot dname(arr) = rdata(nrr) \wedge$

$dname(arr) \in SubDomain(ZtoD(nzone))$

/* ZtoD(z) returns the domain associated with the zone z*/

then

if $(dname(arr), type(arr), class(arr)) \in dom(newrs)$ /* add arr to newrs */

then newrs(dname(arr),type(arr),class(arr)) =

newrs(dname(arr),type(arr),class(arr)) $\cup \{arr\}$;

else newrs(dname(arr),type(arr),class(arr)) = $\{arr\}$;

foreach $nrr \in nss \cdot (rdata(nrr), \mathsf{A}, class(nrr)) \notin newrs \dagger \mathsf{dom}\ Fresh(View(w))$

/* find A rr for servers that do not live in nzone */

dname(temprr)=rdata(nrr); type(temprr)=A; class(temprr)=class(nrr);

tempz = NearestZoneKnown(w,temprr);

temprs = Ans(CheckAuthServer(temprr,tempz,ZtoD(tempz),hop+1,newrs));

foreach $temprr2 \in temprs \cdot dname(temprr) = dname(temprr2) \wedge$

$type(temprr2) = \mathsf{A} \wedge class(temprr2) = class(temprr)$

if $(dname(temprr2), type(temprr2), class(temprr2)) \in \mathsf{dom}\ newrs$

then newrs(dname(temprr2),type(temprr2),class(temprr2)) =

newrs(dname(temprr2),type(temprr2),class(temprr2)) $\cup \{temprr2\}$;

else newrs(dname(temprr2),type(temprr2),class(temprr2)) = $\{temprr2\}$;

Appendix C

Top 100 Web Sites

This appendix contains the domain names corresponding to the "Top 100 Web Sites" selected by PC Magazine in 1998 [49]. These domain names were used in Experiment #1, reported in Section 4.9.3.

```
www.auctionuniverse.com.
www.classifieds2000.com.
www.ebay.com.
www.monster.com.
www.onsale.com.
talk.excite.com.
www.icq.com.
www.talkcity.com.
www.tripod.com.
chat.yahoo.com.
www.buycomp.com.
www.chumbo.com.
www.computers.com.
www.killerapp.com.
www.necx.com.
www.dell.com.
www.macromedia.com.
www.microsoft.com.
www.real.com.
www.3com.com.
www.cnet.com.
```

www.developer.com.
www.oreilly.com.
www.whatis.com.
www.zdnet.com.
www.broadcast.com.
www.imdb.com.
www.mp3.com.
www.mylaunch.com.
www.theonion.com.
www.ajkids.com.
www.discovery.com.
family.disney.com.
www.learn2.com.
www.nationalgeographic.com.
www.etrade.com.
moneycentral.msn.com.
www.quicken.com.
www.thestreet.com.
quote.yahoo.com.
www.gamecenter.com.
www.gamespot.com.
www.ign.com.
www.zone.com.
www.uproar.com.
www.abcnews.com.
www.cnn.com.
www.msnbc.com.
www.nytimes.com.
wire.ap.org.
www.anywho.com.
www.switchboard.com.
www.whowhere.lycos.com.
www.worldpages.com.
www.people.yahoo.com.
www.excite.com.
www.miningco.com.
home.microsoft.com.
home.netscape.com.
www.yahoo.com.
www.bigbook.com.
www.infospace.com.
www.irs.gov.
www.loc.gov.
www.census.gov.

www.av.com.
google.stanford.edu.
www.hotbot.com.
www.metacrawler.com.
www.northernlight.com.
www.amazon.com.
comparenet.com.
carpoint.msn.com.
netmarket.com.
shopping.com.
www.davecentral.com.
download.com.
www.palmpilotgear.com.
hotfiles.zdnet.com.
www.tucows.com.
www.cnnsi.com.
espn.com.
www.golfweb.com.
www.sportingnews.com.
www.usatoday.com.
www.aa.com.
www.biztravel.com.
expedia.msn.com.
www.previewtravel.com.
travelocity.com.
www.devhead.com.
www.gamelan.com.
slashdot.org.
www.webdeveloper.com.
www.w3.org.
www.dejanews.com.
www.hotmail.com.
www.mapquest.com.
www.register.com.
www.webring.org.