University of California

Los Angeles

Improving Internet Resilience through Lightweight Preventive Detection (LPD) and Persistent Detection & Recovery (PDR)

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Lan Wang

2004

The dissertation of Lan Wang is approved.

Deborah Estrin

Songwu Lu

Mani Srivastava

Daniel Massey

Lixia Zhang, Committee Chair

University of California, Los Angeles

2004

To my parents, my husband Qiang and my daughter Serena ...

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am extremely grateful to my advisor Professor Lixia Zhang for her guidance, encouragement and continuous support during my graduate years. She has not only taught me the importance of thinking at a higher level, but also encouraged me to tackle research problems from a different angle. Despite her busy schedule, she has always provided insightful answers to my questions and immediate responses to my concerns. I would like to express my deepest thanks to her for giving me the flexibility of working from my home in Memphis for the past three years. She is my role model of a good researcher and a good mentor.

I would also like to thank Dr. Dan Massey for guiding my work on BGP routing along with Lixia. My discussion with him has always helped me think more clearly. Dan has also provided numerous comments on my previous papers and valuable comments on this dissertation. I would like to express my great appreciation to him for providing me the opportunity to work as a summer intern at ISI East for the past two years.

Thanks to Randy Bush for providing insightful comments on my work. His knowledge on how routing actually works in the Internet helped shape our adaptive path-filter design. I would also like to thank other members of my committee: Professor Deborah Estrin, Professor Songwu Lu, and Professor Mani Srivastava for their valuable feedback on my dissertation.

I have been fortunate to meet many good friends and colleagues at UCLA: Mo Xiao, Beichuan Zhang, Andreas Terzis, Yixin Jin, Xiang Zeng, Mengqiu Wang, Junhong Cui, Dan Pei, Kaixin Xu, Zhiguo Xu, Ken Tang, Nelson Tang and Scott Michel. Special thanks go to my friends Fan Ye, Xiangyu Zhuang, Fang Chu and Zi Yue for helping me solve many telecommunication related problems. I would

also like to thank Sonia Tsui for her friendship and encouragement.

I am deeply indebted to my husband Qiang for his love and support. I would also like to thank my parents for always believing in me. Finally, thanks to my daughter Serena for always cheering me up with her bright smiles.

Vita

1975	Born, Hunan Province, P. R. China.
1993–1997	B.S. (Computer Science), Peking University, Beijing, P. R. China
1997–1999	M.S. (Computer Science), University of Califonia, Los Angeles.
1997–present	Research Assistant, Computer Science Department, University of California, Los Angeles.
2000	Summer Intern, IBM T. J. Watson Research Center, Yorktown Heights, NY
2002	Summer Intern, USC/ISI, Fairfax, VA
2003	Summer Intern, USC/ISI, Fairfax, VA

Publications

M. Gerla, R. Bagrodia, L. Zhang, K. Tang, L. Wang. TCP over Wireless Multihop Protocols: Simulation and Experiments. In *Proceedings of the IEEE International Conference on Communications (ICC'99)*, June 1999

M. Kazantzidis, L. Wang, M. Gerla. On Fairness and Efficiency of Adaptive Audio Application Layers for Multihop Wireless Networks. In *Proceedings of*

IEEE MOMUC 99, Nov. 1999

D. Pei, L. Wang, D. Massey, S. F. Wu, L. Zhang. A Study of Packet Delivery Performance during Routing Convergence. In *Proceedings of the International Conference on Dependable Systems & Networks (DSN'03)*, June 2003

D. Pei, X. Zhao, L. Wang, D. Massey, A. Mankin, S. F. Wu, L. Zhang. Improving BGP Convergence Through Consistency Assertions. In *Proceedings of IEEE INFOCOM 2002*, June 2002

A. Terzis, K. Nikoloudakis, L. Wang, L. Zhang. IRLSim: A General Purpose Packet Level Network Simulator. In *Proceedings of 33rd Annual Simulation Symposium*, April 2000

A. Terzis, J. Ogawa, S. Tsui, L. Wang, L. Zhang. A Prototype Implementation of the Two-Tier Architecture for Differen-tiated Services. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, June 1999

A. Terzis, L. Wang, J. Ogawa, L. Zhang. A Two-Tier Resource Management Model for the Internet. In *Proceedings of Global Internet 99*, Dec. 1999

L. Wang, D. Massey, K. Patel and L. Zhang. FRTR: A Scalable Mechanism to Restore Routing Table Consistency. In *Proceedings of International Conference on Dependable Systems & Network (DSN'04)*, June 2004

L. Wang, A. Terzis, L. Zhang. A New Proposal for RSVP Refreshes. In *Proceedings of the 7th International Conference on Network Protocols (ICNP'99)*, Oct. 1999

L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. F. Wu, L. Zhang. Observation and Analysis of BGP Behavior under Stress. In *Proceedings of the Second ACM SIGCOMM Internet Measurement Workshop (IMW'02)*, Nov. 2002

L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. F. Wu, and L. Zhang. Protecting BGP Routes to Top Level DNS Servers. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, May 2003

L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, L. Zhang. Protecting BGP Routes to Top Level DNS Servers. *IEEE Transactions on Parallel and Distributed Systems,* vol. 14, no. 9, Sept. 2003

X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. F. Wu, L. Zhang. An Analysis of BGP Multiple Origin AS (MOAS) Conflicts. In *Proceedings of the First ACM SIGCOMM Internet Measurement Workshop (IMW'01)*, Nov. 2001

X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. F. Wu, L. Zhang. Detection of Invalid Routing Announcements in the Internet. In *Proceedings of the International Conference on Dependable Systems & Networks (DSN'02)*, June 2002

Abstract of the Dissertation

Improving Internet Resilience through Lightweight Preventive Detection (LPD) and Persistent Detection & Recovery (PDR)

by

Lan Wang

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2004

Professor Lixia Zhang, Chair

It is important for the Internet to function well under adverse conditions caused by operator errors, hardware failures, software flaws, and malicious attacks. As the Internet continues to grow, however, this goal becomes more difficult to achieve since network protocols often need to maintain more state and there are potentially more sources of faults and attacks that can introduce false state information and/or state inconsistencies.

In this dissertation, we propose two efficient approaches to improving network protocols' resilience against network faults and attacks. The first approach, *Lightweight Preventive Detection (LPD)*, exploits network and protocol characteristics to identify protocol messages containing false state information. Because it does not rely on complex cryptographic mechanisms, this approach can produce fast detection mechanisms that are readily deployable in the current Internet. The second approach, *Persistent Detection & Recovery (PDR)*, allows network nodes to periodically correct state inconsistencies without incurring per-state refresh overhead. The key idea behind this approach is state compression – every node

compresses its entire state space into a digest and the digest is used to detect and recover inconsistent state.

We apply the two proposed approaches to protecting BGP (Border Gateway Protocol) against falsely injected routes and undetected routing inconsistencies. Our *Adaptive Path-Filtering* mechanism exploits routing stability and server redundancy to protect important Internet server sites, such as the top-level DNS (Domain Name System) servers, against route-hyjacking. Our *Fast Routing Table Recovery (FRTR)* mechanism allows BGP to recover efficiently from routing table corruptions as well as from transient session failures. Our trace-driven simulation shows that (1) adaptive path-filtering effectively filter out false routes to top-level DNS servers, and it has little negative impact on DNS service accessibility; and (2) FRTR achieves error-free routing tables with a probability close to 100% after only a few rounds of message exchanges, and it reduces the bandwidth overhead of session recovery by one to two orders of magnitude.

Furthermore, we apply the PDR approach to RSVP (ReSerVation Protocol). Our *Scalable Refreshes* mechanism can effectively maintain the state consistency among RSVP nodes while incurring only a constant refresh overhead. Finally, we propose and evaluate two generic state compression techniques, Digest Tree and Bloom Filter-based Digest, which can be incorporated into any PDR mechanism. The effectiveness of these protocol mechanisms and techniques demonstrates that our approaches can lead to a resilient protocol design.

CHAPTER 1

Introduction

The Internet is composed of more than 16,000 Autonomous Systems and it has hundreds of millions of end users. Although Internet users have diverse interests and goals, they all want dependable data delivery services, which in turn requires a resilient infrastructure. A resilient Internet infrastructure should be able to deliver data despite human errors, virus/worm attacks, hardware failures and other problems. The data delivery services may be temporarily degraded when network faults and attacks occur, but they should not be permanently disabled by those events.

Network faults such as human errors pose a major threat to Internet resilience. According to a Yankee Group report [Ker04], 62% of downtime in the surveyed multi-vendor networks is due to human error and the estimated cost of downtime is from $90,000 to $4,500,000 per hour depending on the industry. Human errors have caused failures in both Internet service sites and backbone networks. Oppenheimer et al. studied three large-scale Internet services and discovered that the leading cause of service failures is operator error at two of the three services [OGP03]. A measurement study by Mahajan et al. shows that router misconfiguration is the cause of falsely injected BGP (Border Gateway Protocol [RL95]) routes to 200–1200 address prefixes every day during their observation period [MWA02]. In addition to human errors, software flaws and hardware defects have caused many network outages. For example, in June 2003, a Swedish

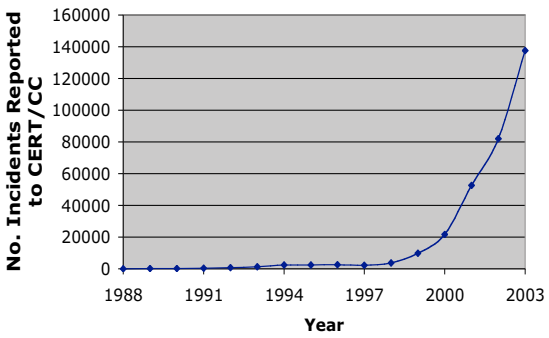


Figure 1.1: Number of Incidents Reported to CERT/CC per Year from 1988 to 2003 (Source: CERT/CC Statistics)

Internet Service Provider had a memory corruption in one of their routers and this single fault caused one million people to lose their Internet service [Don03].

At the same time, we are seeing a rapid growth of malicious attacks. Figure 1.1 shows the number of attack incidents reported to CERT/CC [CER] . It shows clearly that the number of reported incidents has been growing exponentially since the late 90's, rising from 3,734 in 1998 to 137,529 in 2003. In addition, attacks continue to be more automated and more difficult to detect [CER02]. Because regular Internet services are often disrupted by attack traffic and infected machines usually need to be manually restored, network attacks have huge economic consequences. According to Computer Economics, the estimated economic impact was $2.62 billion for the Code-Red attack alone and $13.2 billion for all the network attacks in 2001 [Com].

It should be noted that the original ARPANET was designed with robustness, specifically the ability to withstand fail-stop type of failures, as one of the most important goals [Cla88]. This prioritization of design goals has shaped the architecture of the ARPANET, and later the Internet. In particular, the network provides only a best-effort datagram service and application-specific functions are pushed to the end points (an example of the "end-to-end argument" [SRC84]).

As a result, end points, rather than the network, maintain application-specific state. In this way, the state will expire only when the associated application goes away (i.e. "fate-sharing" [Cla88]). This architectural design choice ensures that application-specific state can survive any link or node failures in the network.

Today we face two major challenges in making the Internet resilient. First, because of the sheer scale and heterogeneity of the Internet, there are many more potential sources of failures. For example, BGP has to deal with a wide variety of faults and attacks that can remove, modify or insert routes (see Section 2.1.1). Our research has also shown that attacks not directed against routers could have an adverse effect on Internet routing because of BGP design and implementation defects (see Chapter 4). Therefore, robustness against only fail-stop type of failures, such as packet losses, node crashes and link failures, is not enough to ensure continued functioning of Internet protocols.

Secondly, we need protection mechanisms that can scale as the Internet grows. The first scaling factor is the amount of state in a protocol. As the number of hosts, routers or Autonomous Systems (AS) increases, many protocols need to maintain more protocol state. For example, there were fewer than 20,000 routes in a BGP routing table ten years ago, but today a default-free BGP table can contain as many as 160,000 routes [Tel]. The second scaling factor is the frequency of message exchanges, which may increase as links become faster and networks become more unstable. A routing protocol, for example, may need to handle thousands of routing updates every minute during unstable periods. Consequently, a scalable protection mechanism should not incur substantial per-state or per-message overhead.

This dissertation therefore seeks answers to the following question: *can we design protocols to be resilient against both existing and potential threats, and at*

the same time, be capable of handling a large amount of state and a high level of message rate? Specifically, a resilient protocol is one that satisfies the following two requirements: (1) it should be able to limit the propagation of false state information; and (2) it should maintain a sufficient level of state consistency among network nodes (see Section 3.3 for more details on these requirements).

1.1 Contributions

We propose two general approaches to improving the resilience of network protocols. Each approach allows a protocol to satisfy one of the aforementioned requirements in a scalable fashion.

Lightweight Preventive Detection (LPD) The goal of this approach is to efficiently detect *false state information* caused by faults and attacks with relatively high accuracy. Rather than relying on complex cryptographic mechanisms, this approach exploits simple network or protocol characteristics to identify suspicious state information. For this reason, it can produce fast detection mechanisms that are readily deployable in the current Internet. When stronger validation mechanisms are available, LPD mechanisms can also serve as a front-end to them by quickly identifying suspicious messages that warrant more inspection. This two-tier design improves the overall efficiency of a protocol.

Persistent Detection & Recovery (PDR) This approach allows network nodes to persistently correct the *state inconsistencies* among them without incurring per-state refresh overhead. The key idea behind this approach is *state compression*—every node compresses its entire state space into a small digest. Instead of sending one refresh message for each state entry, a node sends to its neighbor one digest for all its state entries. The neighbor can then compare lo-

cally stored state entries with the received digest to pin down any inconsistencies. The small digest size allows a protocol to scale to a large amount of state. Furthermore, digest messages are sent periodically to provide continuous protection against any potential state inconsistencies.

To demonstrate the effectiveness of the proposed approaches, we apply them to existing Internet protocols to improve their resilience. Below we briefly describe the specific mechanisms developed for BGP and RSVP.

Adaptive BGP Path-Filtering We propose a Lightweight Detection mechanism called *Adaptive Path-Filtering* for BGP. This mechanism can be used to protect BGP routes to important Internet server sites that have the following two characteristics: (1) BGP routes to the server sites are generally stable; and (2) content on each site is replicated and distributed to several other sites. The first fact enables our path-filtering mechanism to use a heuristic approach to identify potentially valid routes, while the second fact provides tolerance for occasional errors when our mechanism rejects a valid route. The Domain Name System (DNS [Moc87]) has both characteristics required by our mechanism. Moreover, any false routes to the top-level DNS servers can have a disproportionately large impact. Specifically, one false route to a top-level DNS server can increase the DNS response time for millions of Internet users. An attacker can even use a bogus DNS server in conjunction of a false BGP route to inject false DNS responses.

We have evaluated our path-filtering mechanism against one year of archived BGP routes. The trace-driven simulation shows that our mechanism effectively filters out false routes to top-level DNS servers while adapting to long-term route changes caused by topological and policy changes. Furthermore, the path-filtering mechanism has little negative impact on DNS service reachability.

Fast Routing Table Recovery BGP was designed to handle only topo-

logical failures and packet losses. This simple fault model does not take into account software flaws, hardware defects, human errors and malicious attacks that can corrupt a router's BGP table. To allow BGP to recover from the routing inconsistencies resulting from all these faults and attacks, we propose a PDR mechanism called *Fast Routing Table Recovery (FRTR)* for BGP. FRTR uses periodic Bloom-filter digests to detect and recover from any potential routing inconsistency during normal operations. Moreover, FRTR uses digests to recover from transient session resets efficiently – it allows a BGP router to detect which routes have changed and send only those routes instead of its entire table to its peer.

Our simulation results show that FRTR can achieve error-free routing tables with a probability close to 100% after only a few rounds of message exchanges. By comparison, the current BGP approach would not detect any of these errors and even if the errors could be detected, BGP would require a full table exchange to recover. Moreover, FRTR can reduce the bandwidth overhead of session recovery by one to two orders of magnitude, thus speeding up routing convergence following a session reset.

Scalable Refreshes Because RSVP sends periodic refresh messages for every reservation to keep the state of neighboring nodes consistent, its refresh overhead grows linearly with the number of RSVP flows. This overhead has been a major concern for RSVP's deployment in core networks with millions of concurrent flows. Rather than eliminating the periodic refreshes, we apply the PDR approach to RSVP to make it more scalable. Our scalable refresh mechanism replaces all the refresh messages sent between neighboring nodes with a single digest message. Therefore, it achieves a constant refresh overhead regardless of the amount of state in a node.

Based on our work on BGP and RSVP, we propose the following two state compression techniques and provide guidelines on how to select an appropriate state compression technique for a particular protocol.

Digest Tree This technique uses a tree structure to compute a single digest over all the state entries in a node. If any of the state entries between two neighboring nodes is inconsistent, there is a high probability that their digests will not match. The nodes can then start a recovery process by walking down the digest tree until the inconsistent state is identified.

Bloom Filter-based Digest This technique uses Bloom filter [Blo70] to efficiently encode state and uses salted hash functions to remove the false positives inherent in Bloom filter. Different from the first technique, this technique uses a flat structure to compute the digest so that the recovery process is more efficient.

The effectiveness of the proposed protocol mechanisms and techniques demonstrates that our approaches can lead to a resilient protocol design.

1.2 Organization

The remainder of this dissertation is organized as follows. Chapter 2 discusses the limitations of existing approaches to achieving protocol resilience. Chapter 3 presents the requirements for a resilient protocol design. Chapter 4–7 cover our work on improving BGP routing resilience. First, Chapter 4 presents a measurement study of how BGP routing performed during the Nimda worm attack in Sept. 2001. The findings from this study underscore the importance of designing and implementing a protocol to be resilient against *unexpected* failures. Chapter 5 discusses the root causes of existing and potential BGP vulnerabilities. It then presents the challenges in improving BGP resilience and introduces our

solutions. Chapter 6 presents the Adaptive Path-Filter mechanism. Chapter 7 presents the Fast Routing Table Recovery mechanism. Chapter 8 presents the Scalable Refreshes mechanism designed for RSVP. Chapter 9 compares the two state compression techniques used in our work. Chapter 10 discusses the rationale and limitations of the two proposed approaches – LPD and PDR. Finally, Chapter 11 concludes our work and outlines future research directions.

CHAPTER 2

Background and Previous Work

Traditionally, cryptography-based mechanisms have been used to detect messages containing *false state information* [Sch96]. These mechanisms verify the authenticity and the integrity of message exchanges between network nodes, and they typically incur extra computation and communication overhead for *every* message. Therefore, when there is a large amount of state or data shared between the network nodes, high overhead becomes a major barrier to deploying these mechanisms. Another deployment barrier is the lack of an infrastructure for cryptographic key distribution. SBGP [KLS00], for example, suffers from both problems (see Section 2.1.3).

"Soft State" [Cla88] has been used to protect protocols against *state inconsistencies* caused by network faults and attacks. Examples of soft-state protocols include OSPF [Moy98], PIM [DEF94] and RSVP [BZB97]. In a typical soft-state protocol, nodes periodically exchange their latest state information and state that is not refreshed times out. The soft-state refreshes allow state changes to reach a receiver in the presence of message losses. They can also repair state that is corrupted at the receiver's side. Furthermore, the lack of refreshes for a piece of state signals that the state may be obsolete and such signals help the receiver to remove obsolete state in unexpected situations (e.g. the sender may have crashed or rebooted). Despite the many advantages of soft-state refreshes, periodically refreshing *every* state entry is infeasible for a protocol that needs to maintain

a large number of state entries (e.g. consider an RSVP node with one million reservations or a BGP router with 100K routes).

To provide necessary background and highlight the challenges in this dissertation research, we examine previous efforts to improve the resilience of BGP and RSVP in this chapter. We first discuss the limitations of current and proposed BGP protection mechanisms that are mainly cryptography-based. We then look at existing approaches to reducing the soft-state refresh overhead of RSVP.

2.1 BGP Vulnerabilities and Protection Mechanisms

The Internet is composed of many independently managed Autonomous Systems (or ASes). Each AS uses the inter-domain routing protocol BGP (Border Gateway Protocol [RL95]) to calculate the data delivery paths to networks in other ASes. BGP is a path-vector routing protocol. Its route distribution and route selection processes are quite similar to those of distance vector routing protocols. However, in order to prevent the routing loop problem that is inherent in distance-vector routing, BGP attaches a complete path (called an "AS path") to each route. More specifically, a BGP route contains an address prefix for the destination network, an AS path to the address prefix, and a set of path attributes associated with the AS path (e.g. the next-hop router's IP address).

To exchange routing information, two BGP routers first establish a peering session that operates on top of a TCP connection. The routers then exchange their full routing tables in a series of BGP messages. After the initial route exchanges, each router sends only incremental updates for new or modified routes. When a router discovers that it can no longer reach an an address prefix, it sends a message to its peer to withdraw the route. Therefore, a BGP message may

advertise a new route (an *Announcement*), change an existing route (an *Implicit Withdrawal*), or withdraw an existing route (an *Withdrawal*). In addition, BGP also uses periodic keep-alive messages to detect router crashes and link failures.

2.1.1 BGP Vulnerabilities

In general, attacks against BGP can be classified into two categories: outsider attacks and insider attacks ([Mur03] provides a comprehensive analysis of BGP's security vulnerabilities.)

1. Because BGP lacks an adequate authentication mechanism, it is vulnerable to *outsider attacks*. An outside attacker (e.g. man-in-the-middle) can inject a TCP packet with its RST or SYN flag turned on to reset the TCP connection beneath a BGP session [NIS04]. In addition to attacking the TCP connection, the outsider can inject a false BGP message into the *BGP session*. The false message can change the status of the session, modify/remove remove existing routes, or even announce false routes.

2. Because BGP does not have any mechanisms for validating routing information, it is susceptible to false routing information advertised by an *insider*. The insider could be a misconfigured peer or an attacker that has broken into a BGP router.

Note that the outsider attacks require that the attacker be able to "spoof" the TCP connection. More specifically, the attacker needs to guess correctly the 4-tuple information (source address, source port, destination address and destination port) of the TCP connection as well as the sequence number for its bogus TCP packet.

Several previous studies have shown that false routing information does exist in the current Internet routing system and misconfiguration may be a leading cause of such information ([ZPW01], [MWA02]). The following is a summary of their findings.

Zhao et al. studied MOAS (Multiple Original AS) conflicts over a period of 1279 days [ZPW01]. MOAS conflicts arise when two or more ASes announce the same address prefix. Zhao's study found that the number of MOAS conflicts increased from a medium of 683 conflicts a day in 1998 to 1294 conflicts a day in 2001. In addition, most of the conflicts are short-lived, suggesting that they are more likely to be introduced by temporary problems such as misconfiguration than legitimate causes such as multi-homing.

Mahajan et al. analyzed routing updates from 23 route servers over a period of three weeks and discovered that 200–1200 prefixes may be injected into the Internet due to misconfiguration each day [MWA02]). BGP simply accepts all the false routing information because it cannot verify whether an AS is the legitimate originator of a prefix or whether a BGP router can indeed reach a prefix via its advertised route.

In addition to misconfiguration, malicious attacks have also been a source of false routing information. Attackers can use a hijacked network address space to do illegitimate businesses. For example, one network operator reported on the NANOG mailing list that a spammer hijacked a piece of their network using a bogus BGP announcement and used this network address space to originate their spams [Dil98].

Direct attacks against routers have also occurred ([HW01], [CER02]). Attackers have broken into poorly secured routers. They have also launched DoS (Denial-of-Service) attacks against routers by sending a large amount of spoofed

routing packets to the routers [Gil02]. These packets can exhaust link capacity as well as exhaust the victim routers' CPU processing power. Moreover, DoS attacks not directly targeted at routers have been observed to have an adverse effect on the routing infrastructure [COP01].

2.1.2 Existing BGP Protection Mechanisms

The current BGP4 standard [RL95] provides the capability to authenticate messages from a BGP peer. However, this authentication mechanism is not widely used [MP03]. Moreover, the IETF IDR working group is in the process of revising the standard and the authentication capability is to be deprecated in the new specification [RLH03].

The TCP MD5 Signature Option [Hef98] can be used to authenticate a BGP peer and protect BGP against TCP spoofing attacks. This option allows a BGP router to compute, for every outgoing message, an MD5 signature [Riv92] over both the message and a secret key. The peer can identify an injected BGP message by computing a signature over the message using its copy of the secret key – this signature will not match the received signature if the attacker does not know the key. Major router vendors such as Cisco and Juniper support this mechanism.

IPsec [KA98] can also be used to protect the TCP connection between two BGP peers. With IPsec, the peers can encrypt the TCP portion of their messages using a secret key so that it will be extremely difficult for an attacker to inject meaningful BGP messages. Moreover, the attacker cannot even observe the messages sent over the BGP session. Although IPsec provides more protection than the TCP MD5 option, the latter is more widely used [MP03].

In practice, key distribution remains a problem for the above mechanisms.

According to [MP03], the TCP MD5 option is "not ubiquitously deployed at the moment, especially in inter-domain scenarios, largely because of key distribution issues. Most key distribution mechanisms are considered to be too 'heavy' at this point."

2.1.3 Proposed BGP Protection Mechanisms

Several anti-route-spoofing approaches have been proposed previously. The "predecessor" and path finding approach proposed by [SG96, GM95] can be used to verify an AS path, but it cannot prevent an AS from falsely *originating* an address prefix that it cannot reach. The latter problem is addressed by Secure Origin BGP (SoBGP) [Whi04]. For each route, SoBGP verifies two pieces of information. First, it verifies whether the origin AS is authorized to originate the prefix by examining the digital certificate attached to the route. Second, it checks the existence of the AS path using globally distributed adjacency information. Both verification functions require the deployment of a key distribution mechanism.

SoBGP verifies only the existence of an AS path, but a physically viable path may still be invalid due to the routing policies in intermediate ASes. The SBGP protocol proposed by Kent et al. ([KLS00]) addresses this problem as well as a variety of other BGP security risks using a Public Key Infrastructure (PKI). However, this approach calls for significant changes to the current Internet routing infrastructure. It also incurs high transmission, processing and storage overhead. For example, their experiments show that the tested routers can do only 18 validations per second at peak rate [KLM00], which makes it impractical to deploy SBGP in the current Internet.

Goodell et al. proposed IRV, a protocol that provides the Inter-domain Route

Validation service to BGP [GAG03]. To verify an AS path using IRV, a BGP router directly queries the Route Validator in each AS on the path for its routing and policy information. An advantage of this solution is that it does not have to be universally deployed. However, it needs to solve several difficult problems. First, the communication between BGP routers and Route Validators must be secure. Second, each Route Validator needs to maintain a large, up-to-date and secure database. Third, each Route Validator may need to handle the queries from many BGP routers at the same time.

2.2 RSVP Soft-State Refresh Overhead Reduction

The term "soft state" was first coined by Clark [Cla88] [1]. He proposed that routers could maintain flow state that is periodically enforced by end systems through refresh messages. In this way, a router crash will not permanently stop the associated network flows from receiving their desired type of service. This is because subsequent refresh messages will automatically set up the flow state along the new path, and the obsolete state on the old path will eventually expire. In contrast, the hard state approach uses reliably transmitted messages to establish, modify and remove state. State changes are propagated only once after the changes are detected and there is no timer associated with each state entry. In this section, we first discuss the pros and cons of the traditional soft state approach in the context of the RSVP protocol. We then summarize previous efforts to reduce the refresh overhead of RSVP.

[1] Note that soft state had been used by routing protocols such as RIP long before the term was introduced

2.2.1 Functions of Periodic Refreshes

RSVP [BZB97] is a resource reservation protocol that can be used to request specific quality of service for particular data flows across the Internet. RSVP carries such requests to all the nodes along the data path(s) to make the resource reservation. As a *soft-state* protocol, RSVP sets a finite lifetime for all the reservation state. The end points of RSVP data flows maintain their reservation by sending periodic refresh messages along the data paths; a session's state is automatically deleted when its lifetime expires. The periodic Refresh messages in RSVP play the following important roles in assuring correct protocol operation:

1. **Automatic adaptation to route changes**. IP routing changes cause data flows to switch to different paths. By design RSVP refresh messages follow the data paths, thus the first RSVP messages along the new paths will establish the requested reservations, while the state along the old paths is either explicitly torn down or otherwise automatically timed out.

2. **Persistent state synchronization**. RSVP messages are sent as IP datagrams which can be lost on the way. RSVP state at individual nodes may change due to rare or unexpected causes (e.g. undetected bit errors). These factors may lead to momentary inconsistency in RSVP state along the data paths. Periodic refreshes serve as a simple repairing mechanism that correct any and all state inconsistencies in RSVP state for active sessions. Thus the network is free from obsolete or orphaned reservations.

3. **Built-in adaptation mechanism for reservation adjustment**. When either a sender or a receiver needs to change its traffic profile or reservation parameters during a session, it simply puts the modified parameter values in the next refresh message.

2.2.2 Performance Issues with Soft State

The simplicity and robustness of soft state come at a price: the protocol overhead grows linearly with the number of active RSVP sessions. Even in the absence of new control information generated by sources or destinations, an RSVP node sends to its neighboring nodes one message per active sender-session pair per refresh period.

Another performance concern is the reservation setup or tear-down delay caused by occasional losses of RSVP control messages. Although periodic RSVP refreshes eventually recover any previous losses, the recovery delay, which is proportional to the refresh period, can be considered unacceptable in a number of circumstances. One may reduce the recovery delay by reducing the refresh period, however doing so would worsen the refresh overhead problem.

2.2.3 Soft-State Refresh Overhead Reduction

There have been several efforts to address the issues of protocol overhead due to periodic transmission of refresh messages. The scope of the proposed solutions vary widely. At one end of the design spectrum, proposals such as [BGS00] wish to convert RSVP's soft state design to a hard-state protocol with *keep-alive* probes. In [BGS00], instead of refreshing reservation state, neighboring RSVP nodes periodically exchange hello messages. This proposal assumes that the state between neighbor nodes will remain consistent as long as (1) all the messages are reliably delivered once, and (2) no link or node failure is detected. While this proposal avoids the state refresh overhead problem, it fails to preserve the full semantics of RSVP's soft state design with its associated benefits.

Pan proposed sending acknowledgments for trigger messages and using a very

low refresh rate (e.g. once every 15 minutes) after the acknowledgment is received [PS97]. This approach reduces refresh overhead to a certain degree, but state consistency will suffer if attacks or faults occur shortly after a state change is acknowledged. Note that the refresh overhead of this approach still goes up linearly with the amount of state.

Sharma, et. al., suggested setting the refresh timer as a reverse function of the amount of state so that the overall refresh overhead can remain constant [SEF97]. They also proposed allocating more bandwidth for trigger messages than for refresh messages. This "scalable timer" approach is quite similar to the previous one, in that they want to give higher priority to trigger messages and keep refresh rate as low as possible. Unfortunately, both approaches can result in a lower level of state consistency.

CHAPTER 3

Requirements for a Resilient Protocol Design

Network faults and attacks can often create false state in network nodes or make their state inconsistent. Since network nodes rely on their state to perform their tasks, false or inconsistent state information can affect their performance, and, in the extreme case, their ability to perform. In this chapter, we argue that a resilient protocol design should satisfy two basic requirements: (1) it should be able to limit the propagation of false state information; and (2) it should maintain a sufficient level of state consistency among nodes.

This chapter is organized as follows. Section 3.1 presents a formal definition and a classification methodology of protocol state. Section 3.2 then examines how faults and attacks can affect state accuracy and consistency. Section 3.3 presents the two requirements that a protocol should satisfy in order to be resilient against faults and attacks.

3.1 An Overview of Network Protocol State

Distributed nodes usually do not have *a priori* knowledge of their environment, yet their tasks often require such knowledge. For example, a BGP router needs to know its neighbors' best routes in order to compute the best route to a destination. Similarly, a TCP sender needs to know the network loss rate and round-trip time in order to better utilize available network bandwidth while avoiding con-

gestion. State is often used to store the information collected by network nodes. Moreover, network nodes set up state to store critical results computed from the collected information. For example, every BGP router maintains a routing table that indicates how packets should be forwarded to their destination networks. Without its routing table, a router would have to compute a route from scratch every time it receives a packet, which is impractical in a high speed network.

In this section, we first introduce a formal definition of state and classify state into four types. We then use two routing protocols BGP and OSPF (Open Shortest Path First [Moy98]) to illustrate the generality of our state definition and classification methodology. Lastly, we explain why the state consistency among network nodes needs to be maintained continuously.

3.1.1 State Definition and Classification

Let's call a piece of state information a *state entry*. Formally speaking, *a state entry is a variable kept by a node in a distributed environment*. It can be represented by a (key, value) pair, and it is uniquely identified by the key. We classify a state entry into the following four categories:

- **local state**: static or dynamic information originated by the node;
- **learned state**: information learned through communication;
- **computed state**: information derived from local and learned states that is directly used by a protocol to perform its tasks;
- **advertised state**: information advertised to other nodes, which becomes the *learned state* of those nodes.

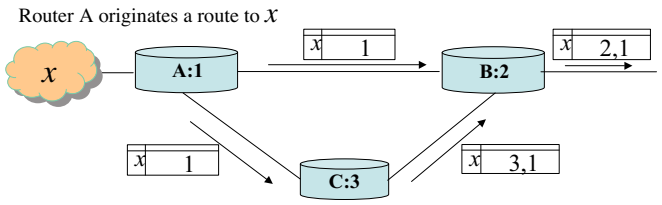


Figure 3.1: Three BGP Routers and Their Routing Updates

To simplify our discussion, we call the node that advertises a state entry a "sender" and the node that maintains the corresponding learned state a "receiver".

3.1.2 Examples of Protocol State

Let's first use BGP to illustrate the above state definition and classification methodology. Each BGP route is a state entry; its key is the destination network's address prefix and its value is a set of attributes such as the AS path and the nexthop router. For each BGP router, its state space is composed of locally originated routes (local states), routes received from its neighbors (learned states), best routes selected from the neighbors' routes (computed states), and routes advertised to the neighbors (advertised states).

Figure 3.1 shows how three BGP routers exchange and compute their routes. The three routers A, B and C are located in Autonomous Systems (AS) 1, 2 and 3 respectively. A is directly connected to network x and this direct route is A's local state. A then advertises to its neighbors B and C the route $\langle x, ASpath(1)\rangle$, which becomes A's advertised state and its neighbors' learned state. C then selects its best route to network x. Since A's route is the only candidate, C selects it as the best route (C's computed state). Next, C appends its own AS number to the selected route and advertises to B the route $\langle x, ASpath(3,1)\rangle$ (C's advertised state). Now B has two routes received from A and C (B's learned

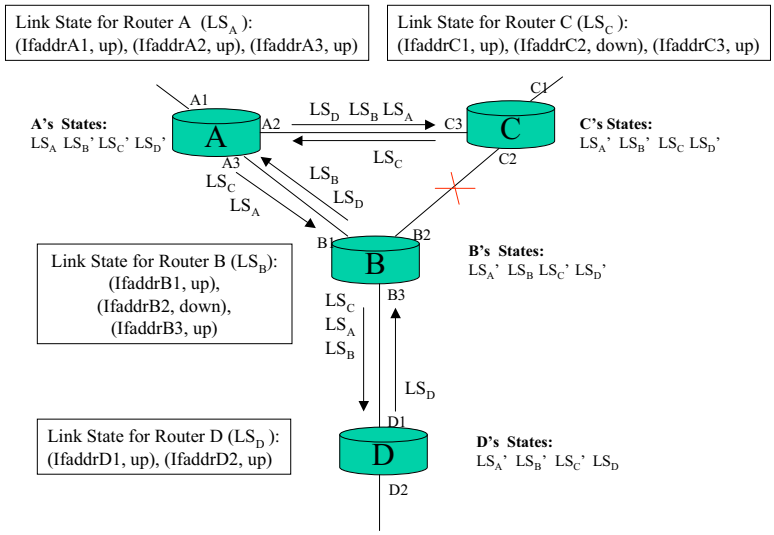


Figure 3.2: Four OSPF Routers and Their Link States

states). It selects the shorter route via A as the best route to network x (B's computed state). It in turn appends its own AS number to the selected route and advertises the resulting route $\langle x, ASpath(2,1)\rangle$ (B's advertised state) to its neighbors.

To show the generality of our definition and classification methodology, we now apply it to the link-state routing protocol OSPF. An OSPF router has the following state entries: the router's own link state, the link states of the other routers in the network and the shortest paths computed from all the link states. A *link state* describes the physical connectivity of a router. It is uniquely identified by the router's address (the key) and its value is the status of the router's links. For example, in Figure 3.2, A's link state LS_A contains information about A's three links whose addresses are *IfaddrA1*, *IfaddrA2*, and *IfaddrA3*. All the three links have the status "up" indicating that they are functional. On the other hand, one of C's links *IfaddrC2* is broken, so the status of that link is "down".

Initially, every router knows only its own link state. They then start flooding their own link state to the whole network (each router also forwards received link

states to other routers). At the same time, they collect each other's link state to build the network topology. Then they compute a route to every network destination based on the topology.

Now we apply the state classification methodology to router A and B in Figure 3.2. The link state LS_A contains information about A's local links, so it is A's local state. Similarly, LS_B is B's local state. A advertises LS_A to B, so it is A's advertised state and B's learned state. To distinguish the copy of LS_A maintained by B from the local state maintained by A, we denote it as LS'_A. A also forwards a copy of C's link state LS_C to B, so it becomes A's advertised state and B's learned state. Finally, the routes computed by router A and B based on the collected link states are their computed states (these routes are not shown in the figure).

3.1.3 State Maintenance

State may change over time. This is one salient difference between state and pure data. Pure data, such as audio streaming data, never change and they are usually consumed (e.g. played out or stored in a file) shortly after they are received. In contrast, the value of a state entry may change at any time and a change in one state entry may trigger a series of changes in other state entries.

Figure 3.3 shows the dependency among different state types. When the value of a *local state* changes, all the *computed states* that are based on the local state should be recomputed. If any of the computed states has a new value, it may lead to state changes in one or more *advertised states*. These state changes need to be communicated to the receiver so that the receiver can update the corresponding *learned states*. Only when the state changes are propagated to the receiver can it adjusts its behavior accordingly. Therefore, in order to perform

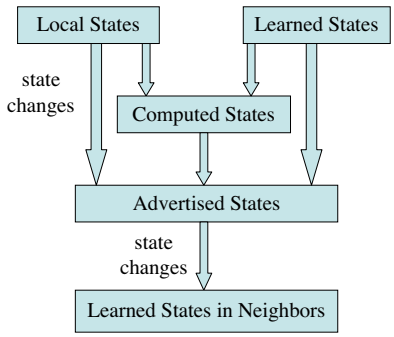


Figure 3.3: Flow of State Changes among Different State Types

their tasks correctly, it is critically important for network nodes to maintain their state consistency. Note that this dissertation focuses on the state consistency *between* neighboring nodes. However, the techniques developed here may be equally applicable to ensuring the state consistency *within* a single node.

Network nodes maintain their state consistency via message exchanges. The following is a description of the typical message exchanges between a sender and a receiver. When the sender has a new state entry s to advertise, it sends a *setup* message to the receiver to install the corresponding learned state s'. Whenever s has a new value, the sender sends an *update* message to the receiver. When s is to be deleted, the sender sends a *teardown* message and the receiver removes s' after receiving the message. The *setup*, *update* and *teardown* message are collectively called *trigger messages*. In the remainder of this dissertation, the term *state change* represents not only the value change in a state entry, but also the creation and removal of the state entry.

3.2 Impact of Faults and Attacks on Protocol State

A misconfiguration can cause a BGP router to leak thousands of routes received from its downstream customer. And an attacker can hijack a popular website

by injecting a false BGP route. Regardless of the details of these faults and attacks, their net effect is *erroneous state information*. Specifically, Figure 3.3 shows that assuming the state consistency within a node can be achieved, false state information can be introduced to a network only when an event accidentally or intentionally changes a local state or a learned state.

Faults and attacks can also cause *state inconsistency* between a sender and a receiver. First, they can change the value of a state entry on the sender thus making the corresponding learned state on the receiver inconsistent. For example, when the link from AT&T to one of its customers (say $NetA$) goes down, AT&T has to withdraw its BGP route to $NetA$. On the other hand, until the withdrawal messages from AT&T are received and processed by the other ISPs, they will still believe that they can reach $NetA$ via AT&T.

Second, faults and attacks may prevent or delay the propagation of state changes, leading to permanent or temporary state inconsistency. For example, some of the withdrawal messages from AT&T may be delayed in certain areas of the Internet due to serious congestion in those areas.

Third, even when the state changes are propagated to the receiver in a timely fashion, the receiver's state may remain inconsistent due to a software bug or a hardware failure. For example, one of AT&T's peers, say Verio, may fail to remove its obsolete route to $NetA$ because a software bug caused it to drop the withdrawal message from AT&T. This kind of failures have indeed occurred before (see [Don99]).

Fourth, even in the absence of any state changes in the sender, the receiver's state may become inconsistent with the sender's, for example, when the receiver's state is corrupted by a hardware failure. Section 5.1 describes a variety of such inconsistencies that can occur between two BGP peers. In our previous example,

a network administrator at Verio may accidentally remove AT&T's route to $NetA$ when he changes the configuration on a BGP router. As a result, even if AT&T's connectivity to its customer is perfect, Verio will not be able to reach $NetA$ via AT&T.

3.3 Requirements

Given the consequences of faults and attacks, it would be desirable to reduce the probability of these state-changing events or make it more difficult for them to modify protocol state. For example, it is important to make router configuration less error-prone to reduce the frequency of BGP misconfigurations. However, a resilient network protocol should still be prepared for any *unexpected* faults or attacks and be able to *minimize* their damage once they occur. In this section, we present two specific requirements for a resilient protocol design.

3.3.1 Limit the Propagation of False State

A resilient protocol should not allow false state information to spread widely. In the case of BGP, this requirement means false routes should be stopped as close to the originator as possible. How well a protocol satisfies this requirement can be measured by the diameter of the propagation range, i.e. the longest path length between the originator and the affected nodes. It can also be measured by the percentage of nodes adopting the false state [ZPW02].

3.3.2 Maintain a Sufficient Level of State Consistency

The protocol should also maintain a sufficient level of state consistency among network nodes *under all circumstances*. The minimum required level of consis-

tency is primarily determined by the requirements of the end users or higher-level protocols. For example, a protocol used in critical missions should in general achieve a higher level of state consistency than a protocol used for recreational purposes. In other words, users have different expectations for these protocols and protocol designers in turn should choose the proper level of consistency.

[RM99] measured *long-term* state consistency using the percentage of time when two nodes have consistent state (i.e. when the advertised state on the sender is consistent with the learned state on the receiver). This measure has also been adopted by [JGK03]. *Short-term* state consistency can be evaluated based on how fast a learned state become consistent with the corresponding advertised state following a state-changing event. A protocol with a high level of short-term state consistency is said to achieve *instantaneous consistency*.

In general, given a certain frequency of state-changing events, increasing the *short-term* consistency of a protocol leads to higher *long-term* consistency. However, if state-changing events occur more frequently, the level of *long-term* consistency may decrease even if the level of *short-term* consistency increases.

CHAPTER 4

Case Study: Internet Routing Behavior during Worm Attack

Before designing a protocol, one may make some assumptions about the typical network environment such as the type of possible faults and attacks. Different assumptions can lead to entirely different approaches, resulting in different levels of state accuracy and consistency. Therefore, whether a protocol is able to meet the two requirements stated in Section 3.3 depends on how accurately the protocol designer's assumptions reflect the reality of the network environment.

One design approach is try to predict all the possible state-changing events before designing a protocol. The core tasks of the protocol then involve the *prompt* detection and handling of every event in this pre-defined set. However, when the reality deviates from the expection of the protocol designer, the protocol's behavior becomes unstable or unpredictable. This kind of problems can occur when network conditions change. For example, DDoS (Distributed Denial-of-Service) attacks used to be rare events and they did not cause much damage, but now they are plaguing the Internet. These attacks can infect thousands of machines in seconds, saturate network links quickly and even lead to failures in critical Internet services.

To illustrate the above problem, we present a study on BGP's behavior during one stressful period, the Nimda worm attack on Sept. 18, 2001. We show

how poorly the current BGP reacts to frequent transient session resets, a failure scenario not envisioned by the original protocol designers. Our study also reveals two other serious weaknesses in BGP's design and implementation. First, BGP is unable to avoid the global propagation of small local changes. Second, the undesirable effects of certain BGP implementation features get amplified under stressful conditions. Our findings have several implications on resilient routing protcol design.

This chapter is organized as follows. Section 4.1 provides more information on the correlation between the Nimda worm and an observed increase in the number of BGP updates. Section 4.2 describes our methodology and system for classifying BGP updates into meaningful groups. Section 4.3 provides a more detailed look at the different classes of updates received during our study period. Section 4.4 presents our major findings. Section 4.5 discusses the implications of the observed BGP problems. Section 4.6 summarizes related work in this area. This chapter is based on our work in [WZP02].

4.1 Nimda Worm's Impact on BGP Update Volume

Nimda is a particularly vicious worm that uses four distinct mechanisms to infect web servers and end hosts [SAN01]. First, it infects Microsoft web servers by exploiting several vulnerabilities in their code as well as the backdoors left by previous worms. Second, it sends itself as an email attachment to addresses harvested from Windows address books, html pages and other sources. If a user or his email software opens the attachment, his computer will be infected by the virus. Third, an infected web server can further propagate the Nimda worm to end users browsing its webpages. Finally, the worm can infect users that are using an open shared folder on a network.

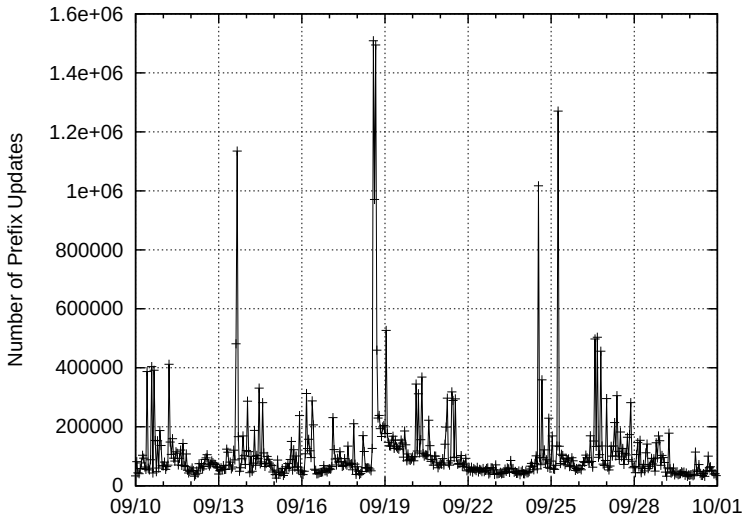


Figure 4.1: Hourly BGP Update Volume (9/10/2001–9/30/2001)

According to the SANS Institute, the scanning activity of the Nimda worm increased dramatically at approximately 1pm on September 18, 2001 [1], and abated in the following hours [SAN01]. Figure 4.1 shows the number of BGP prefix updates received by the RRC00 monitoring point [RIP] at RIPE NCC. One can see a large spike of BGP updates received by the monitoring point around the worm attack time. More specifically, roughly 1.5 million BGP updates were received between 2pm and 3pm, which represents a 30 fold increase over the number of updates received between 12pm and 1pm on the same day. Although large spikes of BGP updates are observed on a few other days as well, the one on Sept. 18 rose much higher and lasted longer. Such behavior was taken as an indication that the worm attack caused global routing instability [COP01].

[1] GMT time is used throughout this thesis.

4.2 Data Source and Methodology

We analyzed the BGP messages collected at RIPE NCC [RIP] from Sept. 10, 2001 to Sept. 30, 2001. RIPE NCC operates several monitoring points and each monitoring point peers with multiple operational BGP routers at various ISPs. Our analysis is based on data from the RRC00 monitoring point located in Amsterdam, Netherlands. RRC00 peers with 15 BGP routers through *multi-hop* eBGP sessions. We analyzed data from 12 sessions that were active during the observation period. Table 4.1 summarizes the locations of these twelve BGP peers. Three of the monitored ISPs are tier-1 ISPs in the US and the others are ISPs in Europe and Asia.

Location	AS Number (ISP)
US	AS7018 (AT&T), AS2914 (Verio), AS3549 (Global Crossing)
Netherlands	AS3333 (RIPE NCC), AS1103 (SURFnet), AS3257 (Tiscali), AS286 (KPNQwest)
Switzerland	AS513 (CERN), AS9177 (Nextra)
Britain	AS3549 (Global Crossing)
Germany	AS13129 (Global Access)
Japan	AS4777 (NSPIXP2)

Table 4.1: RRC00's Active Peers (Sept. 2001)

Before we present our methodology, we would like to clarify the distinction between *a BGP message* and *a BGP prefix update*. A *BGP message* refers to the message used by BGP peers to announce/withdraw a route or to manage the BGP session. In the former case, it can carry one BGP route and *multiple* address prefixes that use the same route in order to minimize transmission

overhead. To analyze the route changes for *individual* prefixes, we studied the sequence obtained by unpacking the BGP messages. These unpacked announcements or withdrawals are referred to as "BGP prefix update" (or "BGP update" for brevity).

4.2.1 Classifying BGP Updates

To better understand the BGP behavior during the attack, we classify all the BGP updates into classes and then infer what may be the leading causes of each class. We are most interested in those that are indicative of actual route changes. Furthermore, we observe whether a behavior (e.g. an increase in a certain class) is specific to a subset of the peers, as such behavior is usually associated with specific BGP implementation features or the ISPs' network characteristics.

To classify the BGP updates, we note the timing of a BGP update and its relationship to the previous update. Useful clues include whether the update follows immediately after a session reset, whether the update follows a route withdrawal, and whether the update contains new route information or a duplicate of the previous information. Based on these clues, we categorize BGP updates into the classes shown in Figure 4.2.

At the top of the class hierarchy are two major classes: *announcements* and *withdrawals*. An announcement contains the sender's BGP route to an address prefix, while a withdrawal indicates that the sender wants to remove a previously announced route.

An announcement can be further classified into three sub-classes. If the sender announces a route to a previously unreachable address, this is a *new announcement*. If the sender announces a route to a currently reachable address and the new route is identical to the current route, this is a *duplicate announcement*.

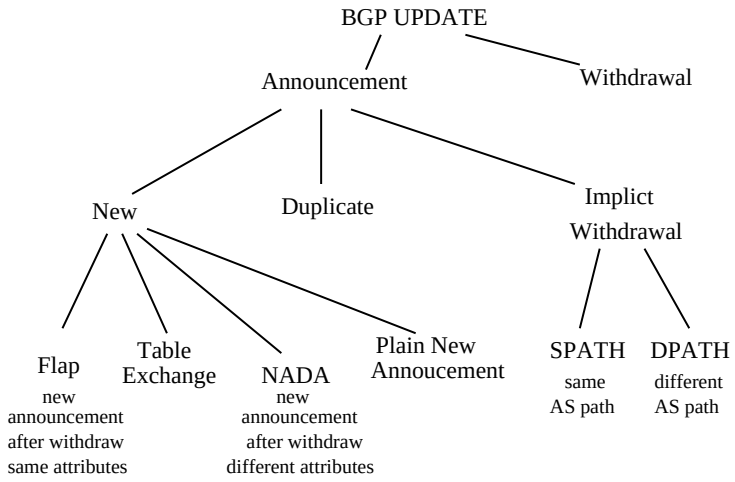


Figure 4.2: BGP Update Class Hierarchy

Otherwise, the sender is replacing the current route with a new route and this is an *implicit withdrawal*.

A *new announcement* can be further classified into four sub-classes. If the new announcement is sent during an initial BGP table exchange, it is labeled as "Table Exchange"; identifying such *Table Exchange* requires special care, as explained in the following section. If it follows a withdrawal and simply re-announces the withdrawn route, it is labeled as "Flap". If it follows a withdrawal and the new route differs from the previously withdrawn route, it is labeled as "NADA". If it fits none of the above three profiles, we call it a "Plain New Announcement".

Finally, an *Implicit Withdrawal* can be further classified into two sub-classes depending on new AS path information. If the new route contains the same AS path as the current route, it is labeled as "SPATH".[2] If the implicit withdrawal replace the current AS path with a new AS path, it is labeled as "DPATH".

[2]An SPATH implicit withdrawal is distinct from a duplicate announcement since the implicit withdrawal changes some attribute other than the AS path.

For example, suppose we want to classify the BGP updates received from ISP1. First, we learn the routes that had already been announced by ISP1 at the beginning of our observation period. We can find this information by obtaining the monitoring point's routing table on Sept. 9, 2001 from RIPE NCC's archive. We then apply each BGP update (collected from Sept. 10 to 30, 2001) from ISP1 to this initial routing table. When a BGP update is received, we categorize it to by comparing the new route with the existing route (in addition to updating the routing table).

4.2.2 Identifying Table Exchange Updates

When a BGP session goes down, the routing table associated with that session is flushed. When the session is re-established, the entire table is re-advertised and the corresponding announcements are counted as *Table Exchange* updates.

Although the log messages in the data set provide the timing information for the peering session state changes, one cannot take a naive approach of classifying all updates received immediately after a session reset as table exchange updates. Because the routing table contains about 100,000 routing entries which may take several minutes to be re-advertised, during this table exchange period other BGP changes may occur and can generate updates that are interleaved with the table exchange updates. Furthermore, there is no clear ending point for the table exchange since the routing table may have added or lost prefixes during the time the session was down.

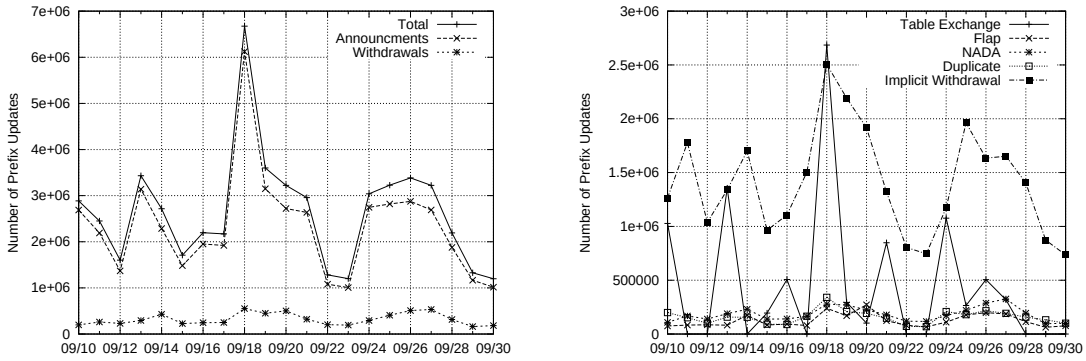
We take a 2-step approach to address this problem. First, based on the observation that most routing table transfers were finished within 10 minutes, we set a relatively long time of 25 minutes as table exchange period. After a new BGP session comes up, our table is initially empty and we count any

update that installs a *new routing entry* as a *Table Exchange* update during the next 25 minutes. Any further updates for this entry are not counted as table exchange updates, even if they occur during the 25-minute table exchange interval. Provided the routing table exchange is finished within 25 minutes, our approach will accurately count table exchange updates with one exception. If a prefix was not in the routing table before the session reset but is announced during the 25 minutes of the table exchange period, a perhaps rare event, it cannot be distinguished from a table exchange update. Thus our count of table exchange updates is equal to the actual number of table exchange updates plus some delta of prefixes that appeared for the first time within 25 minutes of the session reset. If the entire routing table actually took longer than 25 minutes to transfer, also a rare event[3], we would underestimate the total number of table exchange updates.

4.3 Daily Volume of BGP Updates in Various Classes

In this section, we present an overview of the daily BGP update volume from September 10 to September 30, 2001. Figure 4.3(a) shows that the total number of BGP updates varied significantly during the 21-day observation period; the highest value, 6.67 million on Sept. 18 (the worm attack day) is more than 5 times the lowest value, 1.20 million on Sept. 30. The figure also shows clearly that the announcements are the main contributor to the sharp increase observed by RRC00 on Sept. 18. Over the 21-day period the announcements constitute 87.3% of the total number of updates per day on average, and they were even more prevalent on Sept. 18 (91.7%).

[3]We did observe that, in a few cases, the number of prefixes exchanged within 25 minutes of the session reset was much lower than 100K.



(a) (b)

Figure 4.3: Breakdown of Prefix Updates Received by RRC00: (a) Announcements and Withdrawals; (b) Breakdown of Announcements

Further breakdown of the announcements (see Figure 4.3(b)) shows that, except on Sept. 18, implicit withdrawals are the largest component in the BGP announcements, accounting for 40.9% to 81.2% of the total daily announcements. The second largest component is the BGP table exchanges, although this class varies greatly from day to day. There were no BGP table exchanges on 9 of the 21 days, but on the other 12 days they contributed to 3.7% - 43.9% of the total announcements. The combination of these two components caused the other 3 update spikes in Figure 4.1 that occurred on Sept. 14, 24, and 26, respectively. The other three classes, Flap, NADA and Duplicate updates (see definitions in Figure 4.2), are relatively minor contributors to the total count.

Sept. 18 saw both the highest number of BGP table exchange updates – 2.7 million, and the highest number of implicit withdrawals – 2.5 million (see Figure 4.3(b)). The other classes remained insignificant, although they also exhibited increases. More specifically, the composition of the BGP updates on Sept. 18 is as follows (see Figure 4.3): (1) BGP Table Exchanges: 40.2%; (2) Implicit Withdrawals: 37.6%; (3) New Announcements (excluding BGP Table

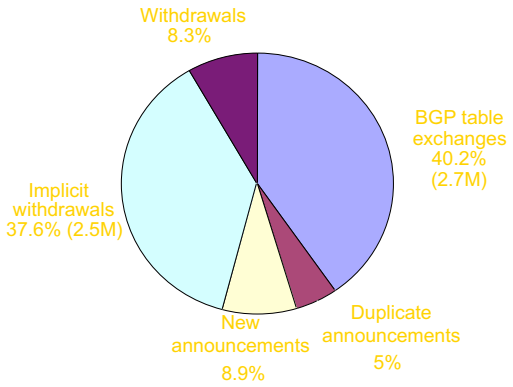


Figure 4.4: BGP Updates in Various Classes on Sept. 18, 2001

Exchanges): 8.9%; (4) Duplicate Announcements: 5%; and (5) Withdrawals: 8.3%. In the following section, we examine the causes of each BGP update class to see whether the increase in that class reflects inter-domain *routing instability*.

4.4 Major Findings

In this section, we identify the major contributors to the dramatic increase of the number of BGP updates during the worm attack. In this process we identify the weaknesses in the BGP protocol design and implementation that gave rise to the observed high routing message volume.

4.4.1 Frequent Session Resets

BGP table exchanges contributed to approximately 2.7 million prefix updates, 40.2% of the BGP updates received by the monitoring point on Sept. 18. If we eliminate this category of BGP updates from the 21-day observation period, the total number of updates received on Sept. 18 is only 1.94 times of the average over the 21-day period (excluding Sept. 18). Below we examine the causes of the large number of BGP table exchanges.

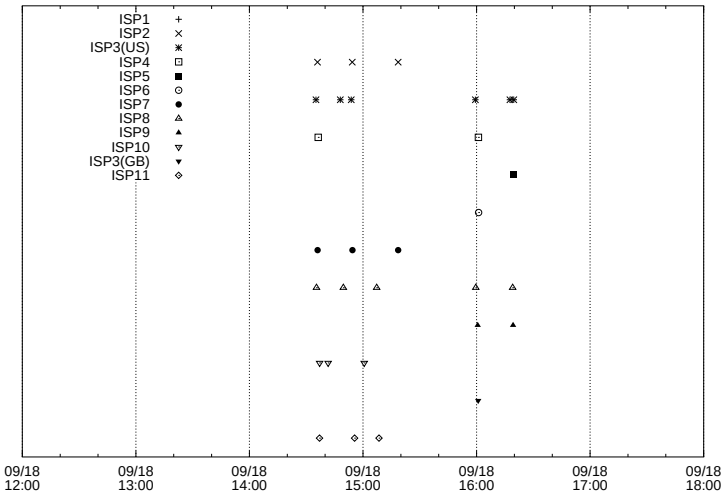


Figure 4.5: Session Resets on Sept. 18, 2001

We first found that, during the worm attack on Sept. 18, 2001, the monitoring point experienced a large number of BGP session resets. Figure 4.5 shows when the session resets occurred on Sept. 18. The X-axis is time and the Y-axis corresponds to the different peers of the monitoring point. Each mark indicates that the corresponding peer's session went down at that time. The figure shows that all the session resets occurred between 2pm and 5pm. In addition, most of the twelve BGP sessions restarted *multiple* times during the 3-hour period and one of them restarted 6 times. Because each session reset means transferring the entire routing table (about 100K entries), even a small number of session resets can result in a large amount of BGP updates.

BGP session resets can be caused by physical connectivity failures (e.g. link failure or router crash), transient connectivity problems due to congestion, or even manual reboots. We observed, however, among the 30 session resets at the monitoring point, 27 were re-established within one minute and the other three recovered within seven minutes. Such fast session recovery suggests that these

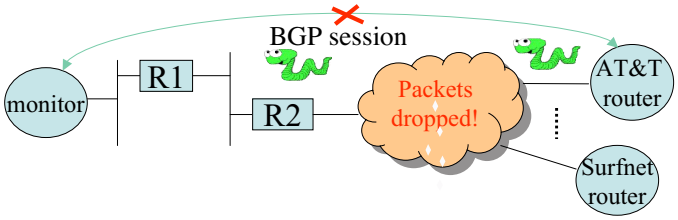


Figure 4.6: Session resets at the monitoring point may be due to the congestion caused by worm scan traffic.

resets were not caused by hardware failures that require human intervention, but most likely were due to transient link congestion or routing problems.

Furthermore, we observed that some of the session resets were highly synchronized. For example, seven session resets occurred between 14:35 and 14:37 and six occurred between 15:59 and 16:03. Using tools provided by the RIPE RIS project to trace the routes from the monitoring point to the peers, we found that the routes to the twelve BGP peers all share the same first two hops, and the second-hop router is one of the BGP routers peering with RRC00. This router had two session resets on Sept. 18, one at 14:36:20 and the other at 16:00:58, an indication of serious congestion or routing problems within the first two hops of the monitoring point at those times. The scan activity of the Nimda worm might have contributed to the congestion that led to the session resets (see Figure 4.6).

However, we would like to remind the reader that RRC00's monitoring peering sessions are multiple-hop eBGP sessions, while peering sessions between operational ISPs are usually set up across high speed LAN or switch interconnect or via direct links. The multi-hop BGP sessions that are commonly used in BGP monitoring projects are likely to suffer from overload at various points along the paths, therefore their *frequent* session resets may not be representative of what happened between the operational ISPs.

At the same time, we would also like to raise the question of whether the BGP

protocol should be designed to work *only* under certain network conditions. An iBGP session may cross several network hops which could be subject to congestion as a multi-hop eBGP session. Moreover, a direct eBGP session could also break due to severe congestion or other types of failures (as you will see in Section 4.4.3). BGP's sensitivity to the transport session reliability raises both a protocol design and an implementation issue that deserve further attention in order to improve the resilience and stability of inter-domain routing.

4.4.2 Causes of Duplicate Announcements

Duplicate announcements make up 4 to 10% of all the BGP updates over the observation period. Although all the peers exhibit this behavior to varying degrees, it is most serious for ISP1, whose duplicate announcements account for, on average, 31% of its total daily prefix updates.

The duplicate announcements may be due to a particular implementation issue [LMJ99]. Such an implementation will send out a BGP update message whenever there is a change to its BGP routes, regardless of whether the change is associated with a *non-transitive* attribute. Since all the non-transitive attributes will be stripped off before a route is announced, the receiver will see a duplicate announcement if all the transitive attributes remain the same. Discussion with ISP1 confirmed the existence of this problem in their router.

It has been argued in the past that tolerating such duplicate announcements leads to a simpler implementation. Our data show that the little saving in the implementation increases the overall system overhead, as all peers have to process additional messages. Worse yet, because changes in non-transitive attributes, such as nexthop and local preference, are usually associated with changes in local network conditions, routers with this implementation feature tend to send

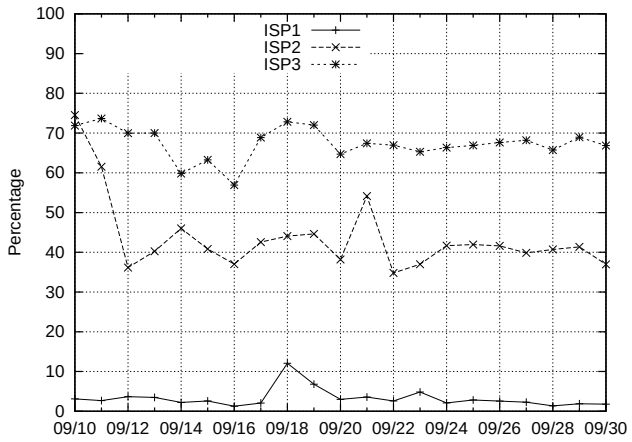


Figure 4.7: SPATH Implicit Withdrawal Percentage (US)

more duplicate updates when their internal network is under stress. Sept. 18 had 0.34 million duplicate updates, the highest over the 21-day period and more than twice the average, suggesting that seemingly harmless small overhead in protocol operation may get amplified under stressful conditions, and that protocol implementation decisions must take into account the potential global impact.

4.4.3 Causes of SPATH Implicit Withdrawals

The real piece of the puzzle is the implicit withdrawals, the second largest component in BGP update counts during the attack period. Further examination shows that on average 22% of all the implicit withdrawals did not contain new AS paths, but involved changes to other BGP attributes such as MED. To distinguish these types of updates, we call the ones without AS path changes *SPATH* and the rest *DPATH*. We examine the SPATH implicit withdrawals in this section and the DPATH implicit withdrawals in the next section.

First of all, we noticed that two US peers had an unexpectedly large portion of implicit withdrawals in this category. Figure 4.7 compares the prevalence of

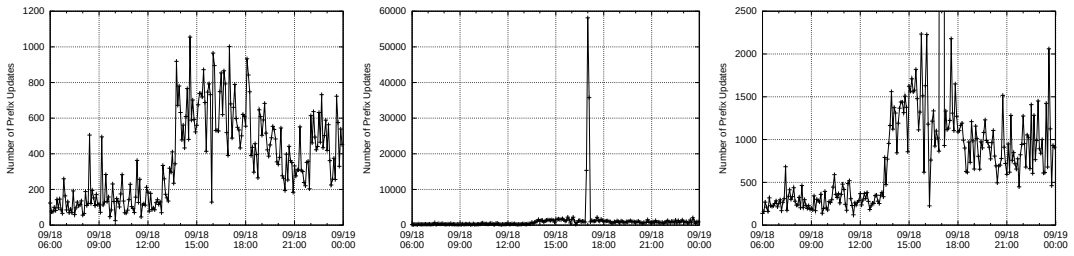
SPATH implicit withdrawals in the three US peers. The X-axis is time. The Y-axis is the percentage of a peer's implicit withdrawals that are SPATH on each day, and we can see that this statistic is about 40% for ISP2 and 70% for ISP3, much higher than the 22% value averaged across all the peers.

Because SPATH implicit withdrawals do not contain new AS paths, they do not reflect topology changes at the AS level, but more likely reflect local changes within an ISP. For example, an ISP may have a policy to dynamically compute the value of the MED attribute (or the community attribute) based on its internal network conditions, as these attributes influence how its neighbors may direct their traffic toward its network. Therefore, the large number of SPATH implicit withdrawals sent by ISP2 ad ISP3 may be explained by the fact that these two tier-1 ISPs have richer network topologies and more sophisticated policies than the other tier-1 or regional ISPs in the monitored set.

The SPATH implicit withdrawals are similar to the duplicate updates in two aspects. First, they do not represent inter-domain routing change. Second, increased *internal* network instability can lead to increased number of SPATH implicit withdrawals. This effect is evident from Figure 4.7, which shows that ISP1's curve rises from a normal level of 2% to more than 10% on Sept. 18 when its internal network was under stress.

4.4.4 Causes of DPATH Implicit Withdrawals

The DPATH implicit withdrawals represent 76.9% of all the implicit withdrawals received on Sept. 18, and they indicate real inter-domain routing changes. In this section we first infer the causes of observed large spikes in the DPATH updates. Then we identify the specific prefixes that contributed most to the DPATH updates. Finally, we observe the impact of BGP slow convergence on



(a) (b) (c)

Figure 4.8: DPATH Implicit Withdrawals: (a) ISP1; (b) ISP5; (c) ISP5 with Spike Removed.

the volume of DPATH updates during the attack period.

4.4.4.1 Resets of Operational BGP Sessions

We plot the updates due to DPATH implicit withdrawals on Sept. 18 for every peer in 5-minute bins. To illustrate the cases we typically see, we select two peers (ISP1 and ISP5) and show their updates in Figure 4.8. ISP1's curve starts to climb around 1pm on Sept. 18 and reaches its peak at 2:35pm (1,055 DPATH implicit withdrawals in 5 minutes). The curve remains relatively high from 2pm to 5pm and then starts to decline. It slowly returns to a normal level after about three days (the figure does not show the days following Sept. 18). However, we can hardly see the same behavior in ISP5 (Figure 4.8(b)), because there is a huge spike reaching 58,150 around 5pm (most points are below 2500). If we remove the big spike by restricting the Y-axis to be between 0 and 2500, we see that ISP5's curve is very similar to ISP1's (Figure4.8(c)).

Based on empirical observations and additional information derived from the routing table, it is plausible to infer that the operational BGP session between the router at ISP5 and one of its peers, let's call it ISPN, went down at this time (see Figure 4.9). As a result, the ISP5 router had to replace all the routes that

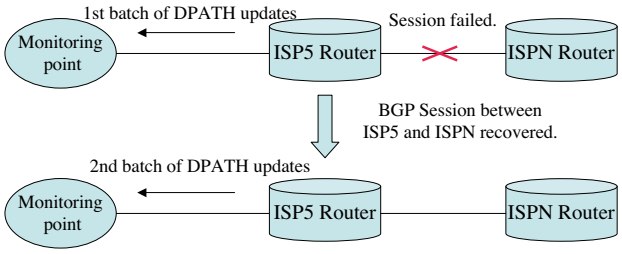


Figure 4.9: Session Failure between ISP5 and ISPN caused a large number of DPATH implicit withdrawals.

used ISPN as the next-hop AS. Because the monitoring point peers with ISP5, we observed all the DPATH implicit withdrawals ISP5 sent out. When the session between ISP5 and ISPN was re-established a few minutes later, all the affected routes had to be restored to their original state, which means another wave of DPATH implicit withdrawals from ISP5 to its peers. We conclude that ISPN was a transit provider and ISP5 used it to reach a large number of prefixes, thus a single BGP session reset between the two could lead to a large number of DPATH implicit withdrawals in a very short time period. The router at ISP1 could also have had BGP session resets with its clients or smaller ISPs but probably did not have any session resets with a major transit peer on Sept. 18, therefore it did not exhibit similar spikes in DPATH implicit withdrawals.

The above behavior indicates that a session reset between two BGP routers may lead to a cascading effect on other routers. Moreover, when BGP session resets involve major carriers and are caused by transient failures, routers adjacent to the involved carriers will typically see *rapid* route changes associated with a *large* number of prefixes over a *short* time period. In fact, we also observed a few spikes in five of the other peers (ISP4, ISP7, ISP8, ISP9, and ISP10) on Sept. 18. Thus we infer that, during the worm attack, some BGP session resets did occur in the operational network relatively close to these monitored routers, although the

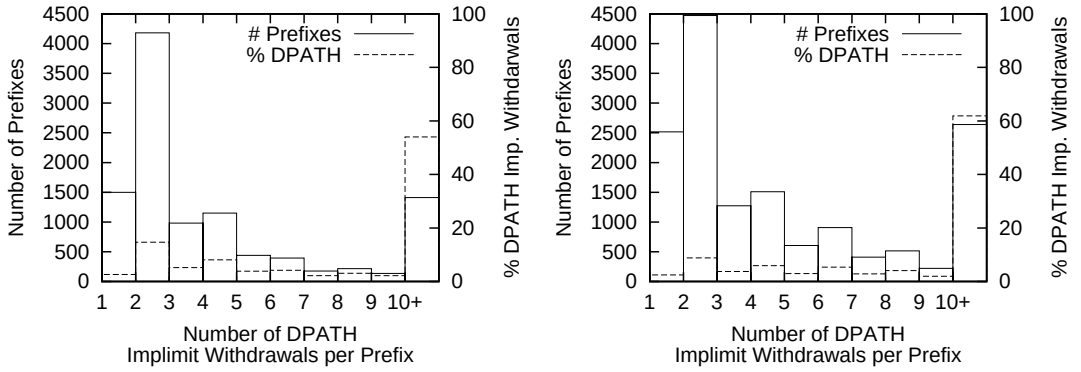


Figure 4.10: Distribution of the Number of DPATH Implicit Withdrawals per Prefix (ISP1): (a) Sept. 17, 2001; (b) Sept. 18, 2001

total number of the resets was probably much lower than that of the monitoring sessions. Note however that even those peers who show no spikes in the number of DPATH implicit withdrawals still have a noticeable increase in their curves. The following two sections examine the causes of these increases.

4.4.4.2 A Small Set of Highly Unstable Prefixes

We first show which prefixes contributed most to the routing changes, by plotting the distribution of prefixes based on the number of DPATH implicit withdrawals each prefix receives on a day. Figure 4.10 compares ISP1's distribution on Sept. 17 and Sept. 18. In both figures, the X-axis is the number of DPATH implicit withdrawals per prefix and we have divided it into ten bins, i.e. 1, 2, ..., 9, 10 and up. The solid line represents how many prefixes fall into each bin and the dashed line represents what percentage of DPATH implicit withdrawals were generated by these prefixes. Their values should be read from the left and right Y-axis respectively. For example, the first column of Figure 4.10(a) indicates that

1,499 prefixes had one DPATH implicit withdraw on Sept. 17 and they account for 2.6% of the total number of DPATH implicit withdrawals ISP1 sent on that day.

Comparing Figure 4.10(a) and (b), we can see that there are more prefixes in all the bins on Sept. 18 than on Sept. 17. This means there were more prefixes involved in routing changes on Sept. 18. But still only 14.4% of the prefixes in ISP1's routing table had at least one route change on Sept. 18.

Let's call the prefixes receiving 10 or more DPATH implicit withdrawals "highly unstable". Figure 4.10 shows that this group of highly unstable prefixes contributed disproportionately to the total count. On Sept. 17, 54% of the DPATH implicit withdrawals were sent for only 1,412 prefixes (1.4% of the routing table). The number in this bin increases to 2,649 on Sept. 18, which means more prefixes became highly unstable. However, they still constitute less than 3% of the routing table and contributed to 61.8% of the total DPATH implicit withdrawals. In fact, all the peers, except one that does not have proper rate limiting, had less than 5,000 highly unstable (see Section 4.4.5), prefixes on Sept. 18, yet this group of prefixes almost always contributed to more than 40% (and sometimes 80%) of all the DPATH implicit withdrawals a peer sent.

We examined 16 prefixes that appeared the most unstable in ISP1's distribution on Sept. 18 (each of them had at least 239 DPATH implicit withdrawals on that day), and found that 13 of them belong to a Cable Modem service provider in the US, two belong to a DSL and dialup service provider in the US, and one belongs to a small service provider in Argentina. Therefore, we suspect most of the highly unstable prefixes during the worm attack are located in edge networks. According to one major router vendor [Cis01], the large number of probes sent by the "Code Red" worm (and similarly Nimda worm) to random IP address caused

Time	AS	Action
09/18/2001 14:04:23	AS3549	originated prefix 66.133.177.0/24
09/18/2001 14:04:37	AS1103	announced aspath 1103 3549
09/18/2001 14:05:10	AS3549	withdrew 66.133.177.0/24
09/18/2001 14:05:36	AS1103	announced aspath 1103 8297 6453 3549
09/18/2001 14:06:34	AS1103	announced aspath 1103 8297 6453 1239 3549
09/18/2001 14:07:02	AS1103	withdrew 66.133.177.0/24

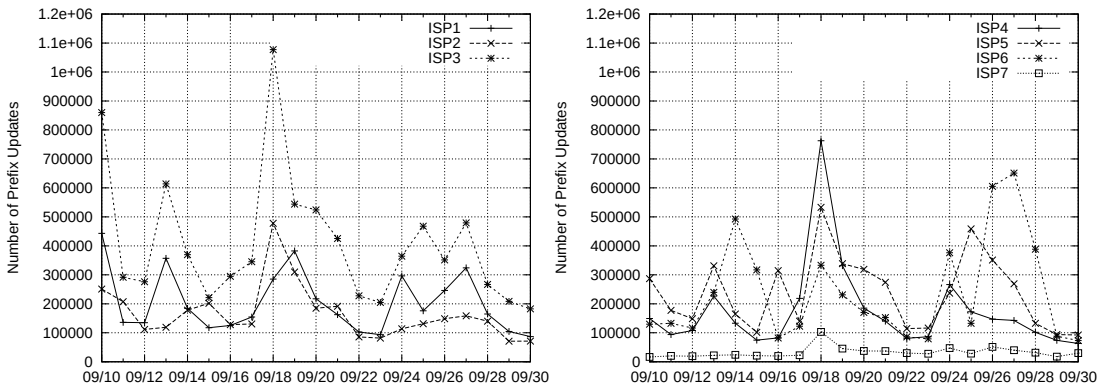
Figure 4.11: BGP Slow Convergence Example

routers in the edge networks to fill up their ARP cache and these routers would restart when their memory is exhausted. The high traffic load also caused some low-end routers to reboot. As a result, the BGP sessions between these networks and their providers would constantly break. These events could have triggered a large number of implicit withdrawals from those providers to their peers. And as we will explain in the next section, the BGP slow convergence problem further amplified the transient instability.

4.4.4.3 BGP Slow Convergence

We believe that a significant number of implicit withdrawals are likely due to the BGP slow convergence problem [LAB00] and they are bogus routing changes. Figure 4.11 shows an example from the BGP updates during the worm attack.

This example shows that prefix 66.133.177.0/24 was withdrawn by its originator AS3549 at 14:05:10 (AS3549 sent a withdrawal message to the monitoring point). We also observed, from the other peers' updates, that AS3549 withdrew this prefix from those peers at the same time, so it is most likely that the connection from AS3549 to the 66.133.177.0/24 network was broken. In reaction to



(a) (b)

Figure 4.12: Comparison among the Peers: (a) US Peers; (b) Peers in Netherlands

the withdrawal message, AS1103 subsequently announced two different AS paths to the prefix. However, both paths were already obsolete because they went through the originator AS3549. The underlying problem is that BGP's path selection algorithm merely tries all the available paths when it receives a withdrawal message, regardless of whether these paths have already been invalidated by the withdrawal message[LAB00]. Such exhaustive search not only results in long convergence time, but also produces a significant number of unnecessary BGP updates. In particular, the withdrawal message in our example triggered a total of 13 implicit withdrawals from six peers, all of which can be avoided if the mechanism proposed in [PZW02] is deployed.

4.4.5 Different Behavior among ISPs

Due to differences in implementations and routing policies, the 12 routers peering with the monitoring point exhibited different behaviors, especially during the worm attack period. As we noted earlier, different ISPs generated different amount of duplicate updates and SPATH implicit withdrawals. Comparing

the *total* volume of updates from different peers, we also found that one of the peer ISPs sent substantially more updates than others. Figure 4.12(a) shows the total numbers of updates by the 3 US peers. We note that ISP3 sent more updates than the other two in general, and more than two times during the attack. Closer inspection reveals that this peer did not exercise proper rate limiting on its updates. For example, the prefix 200.16.216.0/24 was first withdrawn at 00:07:15 on 9/10/2001 and then was announced four times in the following four seconds. This behavior violates the BGP4 specification [RL95] which prohibits the same prefix from being updated multiple times before the expiration of MinRouteAdverTimer (the recommended value for MinRouteAdverTimer is 30 seconds). After consulting the manual of the BGP implementation, we concluded that this behavior is the result of a combination of implementation defect and mis-configuration. The implementation seems benign in normal operations but leading to more pronounced impact under stressful conditions.

Figure 4.12(b) shows that one of the peers in Netherlands (ISP7) sent a much lower number of updates than all the others (note the curve near the bottom of the figure). We determined later that, contrary to the monitoring point's peering agreement, this router was not exchanging the full routing table with RRC00. Therefore, our analysis is likely to have missed certain routing dynamics from ISP7.

4.5 Implications of Observed BGP Problems

Our analysis revealed several weaknesses in the BGP protocol and its implementation. These weaknesses should be addressed to make Internet routing more resilient against unforeseen network faults or attacks. More importantly, future protocol designers and implementors should avoid making the same mistakes that

led to these weaknesses.

4.5.1 Sensitivity to Transport Session Reliability

Over 40% of the observed BGP updates during the attack were caused by BGP session resets at the monitoring point. In addition, several spikes in actual routing changes were probably caused by BGP session resets between operational ISPs. These are evidences that, even though BGP peering sessions seem relatively stable over good connectivity of short distance, they do not work well over "rocky" network connectivities.

The root cause of this problem is that BGP is not prepared to handle frequent session resets. A BGP router needs to exchange with its peer an entire routing table after every session reset, so it performs poorly when its peering session fails frequently under stressful conditions. We believe that a global routing protocol must perform well even under adverse conditions and therefore it should not be engineered against only certain types of failure scenarios.

4.5.2 Amplification of Superfluous Routing Updates

A substantial fraction of the observed BGP updates do not represent AS path changes. Rather, these updates are the results of implementation features that trade performance for convenience or routing policies that expose internal changes within an ISP; the stressful condition caused by the Code-Red/Nimda attack significantly amplified the impact of these seemingly benign choices. This observation underscores the importance of carefully examining any design, implementation or routing policy decision to see if it leads to excessive routing instability in abnormal situations.

4.5.3 Global Propagation of Small Local Changes

The majority of actual routing changes happened to only a small number of highly unstable networks. The intermittent reachability of these networks rippled to the rest of the Internet as rapid BGP update exchanges and BGP's slow convergence made the problem multiple-fold worse. This problem can be mitigated by mechanisms that address the slow convergence issue (e.g. [PZW02]). However, a more fundamental solution is to limit the propagation of BGP updates from extremely unstable sources. Although the BGP route flap damping mechanism [VCG98] was designed for this purpose, our observation indicates that either this mechanism was not widely deployed or it is not effective in damping those sources (or a combination of both). The exact cause is an area of our future research. In general, a truly resilient *global* routing protocol must keep updates generated by unstable or faulty routers as close to their sources as possible.

4.6 Other Studies of BGP Behavior under Stress

Malan and Jahanian [MJ98] observed that BGP sessions would break when their test network was under peak utilization as TCP failed to deliver keep-alive messages in time. Shaikh et. al. [SVK00] also studied BGP's behavior under severe network congestion. Their analytical models and experimental results confirmed that, as the congestion level increases, the expected lifetime of a BGP session decreases. Furthermore, [LAJ99] analyzed the OSPF and BGP updates from a regional network. They showed that BGP's update volume exhibits a daily and weekly frequency while OSPF does not. This finding supports the conjecture that congestion collapse may be a main cause of inter-domain routing failures.

Chang, et al [CGH02] studied the effects of large BGP routing tables on

commercial routers. They demonstrated that some routers would reset one or all of their BGP peering sessions when they run out of memory and then repeat this behavior after they re-establish their BGP sessions. As a result, routing table size would oscillate. When such routers form a chain, the routing table oscillation would propagate. They also studied whether various existing mechanisms can prevent the BGP sessions from failing under large routing table load.

The above studies demonstrate how certain extreme conditions may cause BGP to behave abnormally in both experimental and regional network settings. Our study presents the first in-depth analysis of BGP's behavior under stressful conditions in the *operational* Internet from twelve vantage points.

In a later study, Zhao et al. analyzed the BGP routes to a set of address prefixes belonging to the U. S. Department of Defense (DoD) [ZMW03]. This study showed that the DoD prefixes generated a disproportionately large amount of SPATH updates during the Nimda attack and most of these SPATH updates contain only changes in the AGGREGATOR attributes. The study also revealed that the origin AS (AS568) had multiple BGP routers announce these prefixes and each of the routers attached a different AGGREGATOR attribute to their announcements. Therefore, it is likely that the peers of AS568 were repeatedly switching their next-hop routers during the Nimda attack (possibly due to the highly dynamic network conditions). Since each change in the next-hop router would result in a different AGGREGATOR atribute and this attribute is transitive, the SPATH updates from AS568's peer ASes propagated globally. In summary, this phenomenon is the combined effect of AS568's routing policy, the special characteristic of the AGGREGATOR attribute, and the Nimda worm attack. Such unexpected result again demonstrates that what one normally considers a benign behavior may become harmful under atypical conditions.

CHAPTER 5

Improving BGP Routing Resilience

The original BGP design made several simple assumptions about possible failures in BGP's operational environment and the protocol was engineered to handle only those failures. No further thought was given to how the protocol might behave when the set of failures deviates from the expected set. Unfortunately, as the Internet grows larger and more heterogeneous, those simple assumptions no longer hold. Consequently, false and inconsistent routing information has frequently appeared in BGP tables, causing large-scale routing outages in many cases. In this chapter, we first analyze BGP's design assumptions and their consequences. We then look at the challenges in improving BGP's resilience and introduce our solutions.

5.1 Simple Design Assumptions

5.1.1 Assumption 1: Information from BGP Peers is Valid

Attacks against the routing system was rare when the BGP protocol was designed, therefore BGP provides only a weak authentication mechanism. In fact most BGP sessions are not configured to use this mechanism [MP03]. Even if adequate authentication is available, the problem is that routing information from an authenticated peer could still be false. Unfortunately, BGP assumes that *any*

information from a trusted peer is valid.

Today BGP has to deal with not only malicious attacks, but also human errors. Because BGP does not check the validity of received routes as long as they are syntactically correct, anyone can introduce false routes into the Internet and the effects can propagate globally. To illustrate the severity of this problem, below is a selected list of routing outages caused by false routing information (see [Bon97],[Kro98],[Dan99], [Far01] and [Del02] for more information):

Apr. 25, 1997 At 11:30 am EST, a router in AS7007 received 23,000 routes from a downstream ISP. It accidentally de-aggregated the routes and advertised to its peers 73,000+ routes. A large number of networks became unreachable as a result. This incident was partly aggravated by some BGP implementations' inability to remove the false routes; even after AS7007 disconnected their router, the false routes still persisted for at least seven hours.

Apr. 7, 1998 AS8584 (an ISP in Israel) announced thousands of prefixes belonging to other networks. According to [ZPW01], this incident generated 11,358 MOAS (Multiple Origin AS) conflicts, as viewed from the Oregon Routeviews Server ([rou]).

Apr. 7, 1999 AS7374 leaked many routes via the Internet exchange point CIX (AS1280). It "appears to be announcing most of the Internet" according to a message in the NANOG mailing list.

Apr. 6, 2001 Cable and Wireless (AS3561) had a configuration error that caused it to propagate route announcements from a downstream customer AS15412. [ZPW01] reports that AS15412 normally originated only a few prefixes, but

on this particular day it originated thousands of prefixes, causing at least 5,532 MOAS conflicts.

Oct. 3, 2002 UUNET had an outage affecting millions of its users. The exact cause is unknown, but UUNET reported that the outage was caused by a "route table issue" [MCI02]. According to [Del02], UUNET upgraded the software configuration on a lot of routers at once, but an error in the configuration led to inaccurate routing information.

5.1.2 Assumption 2: Reliable Delivery Can Keep Routing Tables Consistent

BGP assumes that as long as its update messages are reliably delivered by the transport protocol, state consistency between BGP neighbors will be guaranteed. Based on this assumption, it relies on a reliable transport protocol (e.g. TCP) to advertise only *incremental changes* to its routing table, and it does not have the periodic state refresh and timeout mechanisms commonly employed by other routing protocols such as OSPF and RIP.

Past operational experience shows that the above design assumption does not reflect the reality ([Bil98],[Bon97]). For example, on Oct. 25, 1998, an Internet Service Provider (ISP) accidentally sent out a large number of invalid routes, creating an outage over large regions of the Internet [Bil98]. Although the faulty ISP quickly withdrew the false routes, some of the withdrawn routes were still present in certain areas of the Internet on the following day. A similar event occurred on Apr. 25, 1997 in which thousands of accidentally de-aggregated routes persisted in the Internet even after they were withdrawn [Bon97].

The exact cause of the above problems is unknown. However, earlier BGP

implementations by a few major vendors were known to have a common bug that could cause a BGP router to forward a route withdrawal message to some, but not necessarily all, of its neighboring routers [Cha97]. Another software bug could cause a BGP router to drop withdrawal messages before they are processed [Don99]. Since BGP does not delete any route until it is explicitly withdrawn, stale routes persist in the routing table until some external event (such as a peering session failure) flushes out the entire routing table.

In addition to software bugs, hardware failures can also cause routing table inconsistencies. As far back as in the 1970's, a memory corruption of an ARPANET switch caused an east coast router to falsely announce a zero cost route to UCLA [MFR78]. Because ISPs do not normally report all their routing outages or disclose the exact causes, today it is difficult for the research community to gauge how often routing table corruptions occur in the operational Internet. However, several publicized incidents have been discussed on NANOG, the network operators' mailing list (e.g. [Don03]).

Furthermore, malicious attacks can alter routing state [Mur03]. For example, when neighboring routers are connected via a shared medium and their exchanges are not protected by cryptographic mechanisms, another node on the same wire can easily inject a false update. The use of a reliable transport protocol such as TCP does not protect BGP against any of the above *unexpected* faults and attacks.

5.1.3 Assumption 3: Sessions Are Long-lasting and Stable

Based on the assumption that BGP sessions are long-lasting and stable, current BGP design dictates that two peer routers must exchange their entire routing tables after each session reset. Our examination of BGP update logs, however,

shows that BGP monitoring sessions at the RIPE RRC00 monitoring point broke down frequently during the Nimda worm attack in Sept. 2001, possibly due to the transient congestion caused by the worm traffic (see Chapter 4). Moreover, a recent study shows that in the Sprint network, 50% of the failed links recovered in less than one minute [ICM02]. These transient link failures can also cause BGP peering sessions to fail temporarily.

Since a default-free BGP table typically contains over 100,000 routes, a table exchange incurs high bandwidth cost and may delay routing convergence. Such a high recovery cost is particularly unwarranted in the case of a transient session reset – most routes may not need to be retransmitted because they are still valid when the session is re-established.

More seriously, a single BGP table exchange may be able to trigger a vicious cycle of session resets. Specifically, the large volume of BGP messages generated by the table exchange may cause the Keep-alive messages on other BGP sessions to be dropped. As a result, those BGP sessions may reset and start transmitting their BGP tables simultaneously. The resulting higher level of congestion may trigger more sessions to reset, so on and so forth. This vicious cycle of session resets can happen to a BGP router with many peers. For example, a BGP router could have tens or hundreds of iBGP peers. Since these BGP sessions often share a common link, they could all be affected by the congestion on that link.

5.2 Challenges in Improving BGP Routing Resilience

5.2.1 Large Table Size and High Update Rate

Earlier routing protocol designs, such as RIP (Routing Information Protocol [Mal98]) and OSPF (Open Shortest Path First [Moy98]), achieved routing consis-

tency by having routers periodically exchange their latest routes or connectivity information; any information that is not refreshed is deleted. The periodic updates provide protection against a variety of expected and unexpected failures. Unfortunately, due to the large size of today's global routing table, periodically updating every route is infeasible for BGP. For example, as Section 7.6 will show, sending a table with 101K routes would consume nearly 5M bytes of bandwidth. Moreover, there is huge processing overhead associated with sending and receiving these routes.

Mechanisms for detecting false routes must also be scalable with regard to routing table size. In addition, these mechanisms should be able to handle a high rate of BGP updates when routing is unstable. [KLM00] studied the performance of SBGP based on routing table statistics collected in 2000. This study estimated that a router needs around 16Mb to store the signatures received from just one peer. What's more, their routers can validate only 18 signatures a second at peak rate, which means at most 5 updates can be processed every second assuming each update contains on average 3.6 signatures. Apparently, the current SBGP [KLS00] does not meet the requirements on scalability. In general, most cryptography-based mechanisms require intensive computation and are therefore not suitable for real-time processing in high-speed routers.

5.2.2 Incomplete Topology and Hidden Policy Information

Because BGP uses a path vector algorithm, an ISP does not need to disclose its peering relationships or routing policies for other ISPs to compute their routes. However, this benefit comes at an expense – BGP is much more difficult to secure than a link-state protocol (see [Per88] for techniques to secure a link-state protocol).

Unlike routers in link-state routing protocols, BGP routers do not have the *complete* network topology information. As a result, a BGP router cannot verify whether its peer is falsely *originating* a network prefix. Even if BGP is modified so that every router sends its own connectivity information to direct peers, it is still impossible for a router to verify the existence of a *multi-hop* route from its peer to a particular network.

Routing policies further complicate the detection of false routes. Even if a physical route exists between two networks, routing policies set up in the intermediate networks may not allow such a route. Therefore, a router cannot accept routes simply based on network maps produced by network topology inference tools such as Rocketfuel [SMW02].

5.2.3 Inaccurate Authorization Information

Even identifying the correct origin AS of each announcement is challenging since legitimate practices exist that allow one address prefix to be announced by multiple ASes [ZPW01]. In other words, a router cannot consider a received route to be bogus simply because the prefix has already been originated by another AS. Instead, the router needs to know which ASes are authorized to originate the prefix. Although the authorization information is supposed to be available from databases at the Internet Routing Registry (IRR [Mer]), the records in IRR are often incomplete and not up-to-date.

5.3 Proposed Solutions

A resilient inter-domain routing protocol should have at least three tiers of protection mechanisms: (1) a set of lightweight preventive detection mechanisms

for real-time detection of suspicious routes, (2) one or multiple validation mechanisms for narrowing down the suspicious routes to invalid routes, and (3) an efficient mechanism for restoring routing table consistency following transient session resets and other failures. My research has been focused mainly on the first and the third tier. In this section, I summarize my work in these areas (see Chapter 6 and Chapter 7 for more details).

5.3.1 Adaptive BGP Path-Filtering

As we have previously shown, BGP needs additional mechanisms for the detection of false routing information. SBGP is one of the steps in this direction, but it does not scale well. We propose a Lightweight Preventive Detection mechanism called Adaptive BGP Path Filtering. Below I briefly describe how this mechanism is used to protect the Domain Name System (DNS) against route-hijacking.

False routing announcements can deny access to the DNS services. In particular, a single fault or attack against the routes to any of the top level DNS servers can disrupt Internet services to millions of users. Our adaptive path-filtering mechanism protects the routes to the critical top level DNS servers by exploiting two characteristics of the DNS infrastructure. First, there is a high degree of redundancy in top level DNS servers. Second, increasing evidences have shown that most popular destinations, including top level DNS servers, are well connected via stable routes [RWX02]. Our path-filter therefore restricts the potential DNS server route changes to be within a set of stable and verified paths. To adapt to legitimate route changes caused by topology and policy changes, the set of paths in the path-filter is automatically adjusted over time using heuristics derived from routing operations. The high degree of DNS server redundancy provides tolerance for occasional errors when our path-filtering mechanism rejects a

valid route.

We tested the path-filtering mechanism against archived BGP routing updates and the results show that the design can effectively ensure correct routes to top level DNS servers with low impact on DNS service availability. Chapter 6 provides more details about the design and evaluation of our path-filtering mechanism.

5.3.2 Fast Routing Table Recovery

BGP's limited ability to detect routing inconsistencies poses a potential threat to the resilience of the inter-domain routing system. However, one cannot simply re-design BGP to handle a few more failure scenarios that can cause routing inconsistencies. Instead, we argue that it is impossible to enumerate all the potential problems a protocol may face during its operations, and therefore the routing protocol should assume that inconsistencies caused by undetected failures will occur and check the consistency between neighboring routers persistently. However, the large size of BGP's global routing table makes the typical soft-state solution of periodic updates infeasible.

To make BGP more resilient against routing inconsistencies, we propose a mechanism called Fast Routing Table Recovery (FRTR). FRTR can effectively detect and recover from otherwise unnoticeable errors. In FRTR, neighboring BGP routers periodically exchange Bloom-filter digests of their routing state. The digest exchanges not only enable the detection of potential inconsistencies during normal operations, but also speed up routing recovery after a BGP session reset. FRTR achieves low bandwidth overhead by using small digests and is feasible for Internet-scale routing. Moreover, it achieves strong consistency by "salting" the digests with random seeds to remove false-positives.

Our analysis and simulation results show that, with one round of message

exchanges, FRTR can detect and recover over 91% of random errors that the current BGP would have missed with an overhead as low as 1.3% of a full routing table exchange. With salted digests FRTR can detect and recover *all* the errors with high probability after a few rounds of message exchanges. In addition to be a potentially useful mechanism in improving BGP, FRTR demonstrates the feasibility of designing protocols that can be at once resilient, effective, and efficient.

CHAPTER 6

Adaptive BGP Path Filtering

False routing advertisements in BGP have been observed many times over the last few years [MWA02, ZPW01]. Such false routing information could lead to catastrophic results if it prevents Internet users from reaching critical servers such as the top-level DNS (Domain Name System [Moc87]) servers. Our analysis of BGP routing logs from RIPE[RIP] shows that invalid BGP routes to the top-level DNS servers have indeed occurred in the Internet. For example, on April 26, 2001 an ISP incorrectly advertised a route to the "C" gTLD (Generic Top-level Domain) server. A number of large ISPs adopted this false route for several hours; during that time period, all DNS queries to the "C" gTLD server sent by clients in those ISPs were supposedly delivered to the faulty ISP.

Although the top-level DNS servers themselves are closely monitored to guard against compromises, a sophisticated attacker can bypass current security measures by inserting a false network route that redirects DNS queries from an intended DNS server to the attacker. Since secure DNS [ALM02] is not deployed at this time, the lack of authentication in today's DNS service allows the attacker to reply with any desired DNS response. Moreover, even if DNS query replies are authenticated, hijacking the DNS server routes could still lead to denial-of-service attacks. Thus even a single false route announcement for a top-level DNS server can have catastrophic consequences impacting millions of Internet hosts.

We propose a path-filtering mechanism to protect the universally critical BGP

routes that lead to the root/gTLD DNS servers. Our design makes use of two basic facts. First, increasing evidences show that popular destinations have relatively stable routes [RWX02]. Our analysis also shows that the root/gTLD DNS servers can be reached through a set of stable routes (see Section 6.2). Second, the root/gTLD DNS servers have a high level of redundancy, therefore the DNS service will not be impacted by a temporary loss of reachability to one of the servers. The first fact enables our path-filtering mechanism to use a heuristic approach to identify potentially valid routes, while the second fact provides tolerance for occasional errors when path-filtering rejects a valid route. We have tested our path-filtering design against 12 months of BGP routing logs. Our results show that, after applying our path filter, root/gTLD server reachability remains high while invalid routes are successfully blocked by the filter.

This chapter is organized as follows. Section 6.1 analyzes the impact of false routes on DNS. Section 6.2 examines the stability of root/gTLD server routes. Section 6.3 describes a simple path filter example for root/gTLD server routes protection. Section 6.4 presents our adaptive path filter design and Section 6.5 evaluates our design. Section 6.6 summarizes the contributions of this work. This chapter is based on [WZP03].

6.1 Impact of False BGP Routes on DNS

The Domain Name System [Moc87] is an essential part of the Internet infrastructure. It provides the service of translating host names, such as www.cs.ucla.edu, into IP addresses that are used for data delivery. The DNS name space is organized in a tree structure, as shown in Fig. 6.1. A DNS resolver requests data by first querying one of the root servers. Using the referral information from the root server, the resolver proceeds down the tree until the desired data is obtained. For

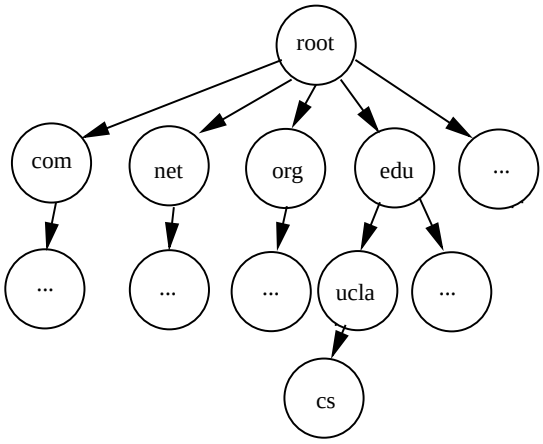


Figure 6.1: The DNS Name Space Tree Structure

example, a resolver seeking the IP address of *www.ucla.edu* starts by querying any root server and the root server provides a referral to the DNS servers for the *edu* domain. A query to any of the *edu* DNS servers returns a referral to the *ucla.edu* servers and finally a query to one of the *ucla.edu* servers returns the IP address for *www.ucla.edu*. In practice the process is slightly more complex; a resolver typically queries a local DNS server and the local server performs the query starting from the root.

To both protect against faults and help distribute the load, each zone in the DNS tree should operate multiple DNS servers in diverse parts of the network. For the root zone, there are 13 root servers and each has an identical copy of the DNS root zone[1]. There are also 13 gTLD servers that serve three top-level domains com, net and org. If one server fails to reply, the resolver simply tries the other replicate servers.

A recent ICANN report suggests that the DNS system can continue to operate correctly even when 5 of 13 root servers are unavailable [ICA02]. However, the

[1]Size limitations in the DNS protocol restrict the maximum number of root servers to 13.

report overlooked the impact that network routing faults or attacks might have on the reachability to the root servers. Hijacking DNS queries to even a single DNS root/gTLD server can have catastrophic consequences. The DNS service, as currently deployed, has no authentication mechanism other than a query ID number chosen by the resolver sending the query. Any response with a matching query ID is accepted. Our goal in this study is to add a simple protection mechanism to guard the DNS service against faults and attacks through false routing announcements.

6.2 Stability of Top-level DNS Server Routes

In this section, we analyze BGP log data to verify one important design assumption: the stability of the BGP routes to the root/gTLD DNS servers.

6.2.1 Data Source

Our data set contains BGP updates collected by the RRC00 monitoring point at RIPE NCC [RIP] from Feb. 24, 2001 to Feb. 24, 2002. We choose this monitoring point because its peers export their full routing tables without imposing routing policies (other monitoring points maintained by RIPE NCC and RouteViews [rou] do not have this peering policy). Nine peer routers stayed active during the one-year observation period (see Table 6.1). Some of them belong to large global ISPs and others belong to regional ISPs. Before conducting the analysis, we carefully removed the known measurement artifacts from the collected data using the technique described in Section 4.2.2.

For each DNS root/gTLD server, we identified the longest address prefix that covers the server's IP address and extracted the BGP updates for these prefixes

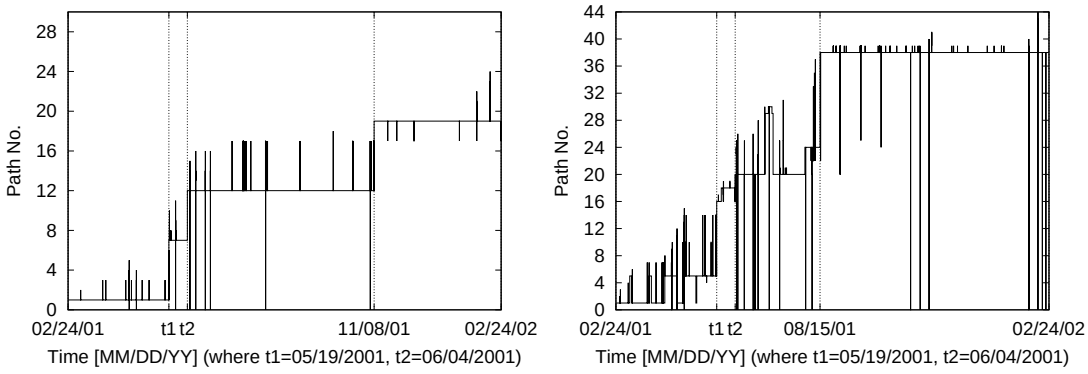
Location	AS Number (ISP)
US	AS7018 (AT&T), AS2914 (Verio)
Netherlands	AS3333 (RIPE NCC), AS1103 (SURFnet), AS3257 (Tiscali)
Switzerland	AS513 (CERN), AS9177 (Nextra)
Britain	AS3549 (Global Crossing)
Japan	AS4777 (NSPIXP2)

Table 6.1: RRC00's Active Peers (Feb. 24, 2001 – Feb. 24, 2002)

from the archive. It should be noted that the "A" and "J" root servers share the same address prefix 198.41.0.0/24 in the BGP routing table. The IP addresses of several gTLD servers changed during our study period. We learned from the operators of those gTLD servers that, during the address transition, the gTLD servers in question were reachable through both the old and new addresses for up to 3 days to ensure DNS service reliability. Therefore, for each gTLD IP address that changed, we used a 3-day overlapping period during which BGP updates for both the old and the new prefix would be included in our data set.

6.2.2 Routing Stability

Fig. 6.2 shows ISP1 and ISP2's AS paths to the A root server. The x-axis is time and y-axis is a unique number assigned to each specific AS path. For example, (x_1, y_1) indicates that path y_1 is being used at time x_1, and a shift from (x_2, y_1) to (x_2, y_2) means the path to reach the A root server changed from y_1 to y_2 at time x_2. In addition, a path labeled 0 means there is no route to the A root server. The figures show that, during this one-year period, each of the two ISPs used a small number of primary AS paths to reach the A root server. In particular, the figure shows that one primary path was used most of the time;



(a) (b)

Figure 6.2: AS Paths to the A Root Server: (a) ISP1; and (b) ISP2

when the primary path was unavailable, a few alternative paths were used for very short time periods. Moreover, the primary AS path usually lasted weeks or even months before the ISP switched to a new primary path. The other 7 ISPs peering with the monitoring point also show similar stability in their routes to the root/gTLD servers; their figures were omitted for brevity.

To illustrate how the characteristics of the routes to reach the DNS servers shaped our design, we present ISP1's AS path changes to the A root server in more detail. The 3 dotted lines in Fig. 6.2(a) divide the graph into four regions; a different primary path is used in each region. The first region begins on February 24, 2001 and lasts until May 19, 2001. During this period, ISP1 used path 1, $(7018, 4200, 2645)$, to reach the A root server except for 1.7 hours when ISP1 used a few other paths. The figure also shows two instances when ISP1 had no path to reach the A root server: a 28-second period on April 16, 2001 and a 157-second period on April 22, 2001.

On May 19, 2001, ISP1's primary AS path to the A root server changed from $(7018, 4200, 6245)$ to $(7018, 10913, 10913, 10913, 10913, 11840)$; the old path never reappeared. The origin AS for the A root server's address prefix was

changed from AS6245 to AS11840, both AS6245 and AS11840 are owned by the same organization which runs the A root server. This origin AS change reflected an operational change. The AS path $(7018, 10913, 10913, 10913, 10913, 11840)$ became the new primary path and it exhibited the same stable behavior as the previous one, being used almost all the time till June 4, 2001. During the few occasions when some alternative route was used, none of them lasted more than 15 minutes.

On June 4, 2001, the origin AS for the A root server changed again and $(7018, 10913, 10913, 10913, 19836)$ became the new primary path. Again this change reflected an operational change by the site running the A root server. A third change occurred on November 8, 2001 when a transit AS, AS 10913, changed its routing policy. AS10913 stopped listing its own AS number multiple times and the new primary path became $(7018, 10913, 19836)$.

In summary, the primary AS path to the A root server changed 3 times over the 12-month period, twice because of a change in the origin AS and once because a transit AS changed its routing policy. Assuming some means existed to adjust the path filter following these changes, any router peering with ISP1 could apply a simple filter containing the primary AS path to protect its route to the A root server. Its reachability to the A root server might be impacted slightly, but it would have gained the protection against any invalid routes to the A root server.

6.3 A Simple One-Path Filter

To illustrate the protection power of path-filtering, we first consider using a *single* route as the filter. In other words, we assume that an ISP selects one allowable AS path for each of the 13 DNS root/gTLD servers and rejects any other paths

it receives from BGP route advertisements. Because a high level of redundancy is built into the root/gTLD server implementations, the simple filter's lack of adaptability to transient route changes does not necessarily impact DNS service availability.

As we have seen in the previous section, network topology and routing policies do change over time and can lead to long term changes to the primary paths for the top level DNS servers. Therefore, although the one-path filter does not adapt to transient route changes, it must adapt to long-term changes. Note that the DNS root servers' IP addresses may also change, but the changes are very rare as they affect millions of hosts. Such changes are widely announced ahead of time, which gives ISPs enough time to adjust their filter setting.

In the rest of this section, we estimate an upper bound of the one-path filter's performance assuming that ISPs are able to adjust their filters to accommodate long term routing changes at the right time. More specifically, we divided the 12-month data into week-long segments and, for each week, we set the filter to be the primary AS path of that week.

6.3.1 Root Server Reachability

We applied the one-path filter to each ISP's BGP updates. Because these BGP updates contain the routes that an ISP chose to advertise to its peers rather than the routes that the ISP received from its peers, we are in effect simulating the filter's effect on a BGP router which has the ISP as its single peer (e.g. one of the ISP's single-homed customers). Due to space constraint, we present the results for ISP1 (the results from the other ISPs are similar).

The graph in Fig. 6.3 shows the effect of the filter on the simulated router's overall reachability to the 13 root servers. Before applying the filter, the router

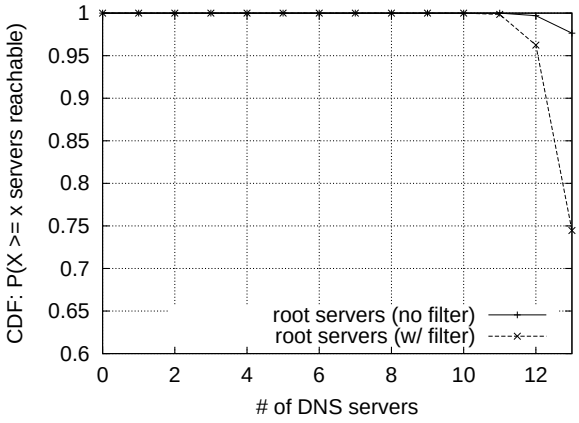


Figure 6.3: Overall Root Server Reachability

has paths to reach all the 13 servers about 98% of the time, and can reach at least 12 servers nearly 100% of the time. After applying the filter, it has paths to all the 13 servers about 77% of the time, and can reach at least 12 servers about 96% of the time. Most importantly, it can always reach at least one server with the filter. In summary, the results indicate that reachability can be extremely high if the filter can be adjusted according to long-term path changes.

After a topology or routing policy change, BGP explores the (potentially very large) set of all possible routes before converging on a new stable route or declares the prefix is unreachable. Previous measurement [LAB00, PZW02] showed that this convergence delay may last 3 minutes on average, and some non-trivial percentage of cases lasted up to 15 minutes. The transient paths during the slow convergence period do not necessarily provide reachability to the destination.

Let us look at a specific example. On April 16, 2001, ISP1's primary path to the A root server failed and was replaced by a back-up path. This back-up path was in place for only 103 seconds before it was withdrawn, and the A root server was declared unreachable. One plausible explanation for this behavior is that

some link near the A root server failed and the link failure invalidated both the primary and the backup path. If this explanation is indeed the case, as we believe it is, then rejecting the backup route did not decrease the actual reachability to the A root server.

We examined the BGP updates to determine how many back-up routes might have been the result of BGP slow convergence. Between Feb. 24, 2001 and May 19, 2001, there were 15 times when ISP1's primary AS path to the A root server was replaced by a back-up route. The majority of the back-up routes remained in place for only seconds. In only 6 out of the 15 instances the back-up route remained in place for longer than 3 minutes, and only 1 back-up route lasted longer than 15 minutes. With our limited data set, we could not confirm which routes were valid back-up routes and which were simply an artifact of BGP slow convergence. However we speculate that those short-lived back-up routes are most likely invalid ones. By rejecting these false back-up routes, the one-path filter would not decrease the actual reachability to the root server and could actually help stop the propagation of transient routes.

6.3.2 Limitations of the One-Path Filter

Overall, the ISPs examined in our study seem to use a single primary path to reach each top level DNS server. Thus if a router peering with any of the ISPs had applied the simple path filter as a protection measure, it would have maintained a high level of reachability to the DNS root servers. However, one of the ISPs performed much worse than the others. If a router peers with this ISP, then the reachability to *all* the 13 root servers is nearly 100% of the time without the filter, but drops to only 35% of the time after applying the filter. Although the router could still reach at least 1 root server 99.97% of the time, the decrease in

reachable servers raises a concern. By analyzing the BGP log data we observed that this ISP uses one primary path and a small number of consistent back-up paths to reach some of the root servers. This fact suggests that, in addition to trusting a primary path, we must enhance our one-path filter with several allowable back-up paths to provide both strong route protection and high service availability.

As Section 6.2.2 shows, updates to the primary path are also needed from time to time. An ideal filter would automatically detect the primary path changes and update the filter accordingly. In the next section we extend the one-path filter to a design that can support multiple valid paths as well as automated filter updates.

6.4 Adaptive Path Filter Design

The simple one-path filter is effective in filtering out invalid paths, but it allows only a single route to each root/gTLD server and requires occasional manual updates when the primary AS path changes. In this section, we present a more advanced filter design that maintains a set of potentially valid paths for each root/gTLD server and automatically includes new valid routes that result from topology or routing policy changes. We use both route history and external validation mechanisms to identify new valid paths.

6.4.1 Design Overview

Our path filter has three components (see Fig. 6.4).

- A *monitoring process* is used to identify potential new paths and keep track of route history.

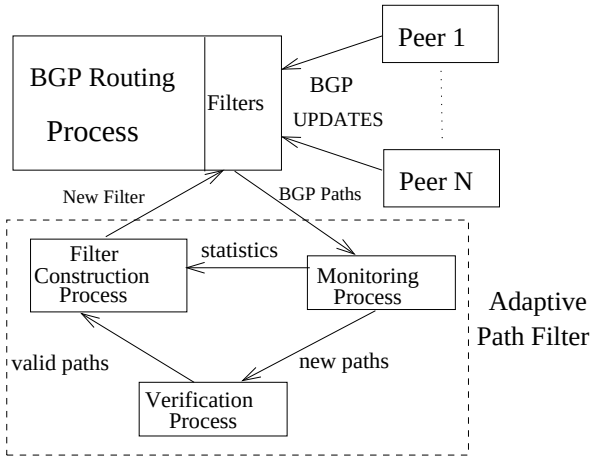


Figure 6.4: Adaptive Path Filter Design

- A *verification process* is used to validate the new path and periodically re-verify existing paths.

- A *filter construction process* dynamically adjusts the filter for a root/gTLD server based on the BGP path statistics collected by the monitoring process and the feedback from the verification process.

To set up a path filter for a DNS server, we first identify the address prefix that covers the IP address of the server, and then choose a set of initial paths that can be used to reach the server. This initial choice is based on past experience and known routing policies. The filter algorithm then adjusts path filters at time intervals of length T in order to accommodate dynamic changes in topology and routing policy.

During each time period, the router monitors the percentage of time a path is announced by a peer, regardless of whether the path is currently in a filter. At the end of a time period, we consider only paths that have been announced more often than a configured threshold. The threshold is used to screen out misconfigured and transient paths; those paths whose usage exceeds the threshold are called

base paths. The verification process is then invoked to check whether the base paths are indeed valid. All previously adopted paths are also verified again with a probability P_v. Only paths that have appeared as base paths and that have passed the verification test will be placed in the filter set for the next time period.

If a new legitimate route appears at the beginning of a time period, the route could be delayed for T amount of time before being added to filter. Therefore, as an enhancement to the above algorithm, we also adopt *valid* paths into the path filters if they have been advertised for a period of T_r in the current time period. T_r is typically much smaller than T and, an appropriate value of T_r should ensure quick adaptation to topology changes while minimizing the impact of transient paths. These paths also need to be validated by the verification process before they can be added to the filter set. If these paths do not meet the criteria for base paths, they will be eliminated from the path filters at the end of the time period T. Sections 6.4.2–6.4.4 describe the algorithm in more detail.

6.4.2 Monitoring Process

The monitoring process collects path statistics. In the $k'th$ time period, it keeps track of $T_k(p)$, the amount of time each path p is used by a peer to reach the DNS server D. At the end of the time period, $T_k(p)$ will be used by the filter construction process to construct the filter set for the next time period.

In addition, if p is not in the current filter set and if $T_k(p)$ exceeds a threshold T_r, then the verification process is invoked to determine whether p should be added to the current filter set. If the verification process determines that the path is valid, p is immediately placed in the filter set.

```
while in_current_time_period()
        %Receive new paths from verification process
        p = recv_new_path();
        F = F ⋃ p;
        p.checked = 1;
end
%Adjust F at the end of a time period
F_old = F;
F = ∅;
foreach p ∈ F_old
   if U(p) ≥ U_min ∨ U_k(p) ≥ U_min
      %Consider only base paths
      then
            if p.checked = 1
               then
                     %p is already validated
                     F = F ⋃ p;
                else
                     %Validate p with a probability of P_v
                     if rand() ≥ P_v ∨ verify(p) = 1
                        then F = F ⋃ p;
                     fi
            fi
   fi
   p.checked = 0;
end
```

Figure 6.5: Algorithm for Filter Adjustment

6.4.3 Filter Construction Process

At the end of each time period, a new filter set is constructed based on the old filter according to the following steps. Fig. 6.5 shows the algorithm in pseudo code.

First, each path's usage is calculated as $U_k(p) = T_k(p)/T$ where $T_k(p)$ is the value provided by the monitoring process and T is the length of the time period. Paths with a $U_k(p)$ above a minimum threshold will be considered as base paths and are candidates for inclusion in the next filter set. However, if a path is oscillating between extreme values of $U_k(p)$, then creating the new filter set based solely on this measure can cause a valid path to be moved in and

out of the filter set at the wrong times. To account for this problem, the filter construction also uses $U(p)$, an exponentially weighted moving average (EWMA) of $U_k(p)$, to select base paths. $U(p)$ is calculated using the following formula

$$U(p) = (1 - \alpha) * U(p) + \alpha * U_k(p), \tag{6.1}$$

where $\alpha \in (0, 1)$.

A path is considered as a base path only if $U(p)$ or $U_k(p)$ is greater than the minimum threshold U_{min}. For any new base path, the verification process will be invoked. For base paths that have been verified in the previous intervals, the verification process is invoked with a probability of P_v. Any base path that fails the verification process is rejected and the remaining base paths form the new filter set.

6.4.4 Verification Process

Verification can be performed by either humans or automated programs. The separation of the monitoring and verification functionality in our design allows each site to experiment with different approaches to verification and improve them based on experience. The simplest verification method is to inspect the BGP path for anomalies such as reserved AS numbers and perhaps use the IRR (Internet Routing Registry) to verify if the origin AS is authorized to originate the prefix [Yu]. However, this method is not reliable because the BGP path may not contain obvious anomalies and the records in IRR are often inaccurate. [BBL98] proposed a DNS-based method to verify the origin AS of a prefix. SBGP ([KLS00]) and a variety of other techniques ([ZPW02, GAG03]) can also be used to verify the validity of a BGP path. Moreover, DNSSEC ([MR01]) can verify the authenticity of a DNS server.

We propose an automated verification process that utilizes the redundancy of the DNS system. The basic idea is to send a DNS request to the DNS server on the suspected path and validate the reply against answers from other DNS servers for the same request. Note that, to ensure that the request will travel on the suspected path, we may execute the verification process only when the peer is still using the path. If the suspected path is injected accidentally, no reply may be received. Otherwise, a reply may come from an impostor (i.e. a false DNS server set up by the attacker who injected the suspected path) and contradict that of the other DNS servers.

The impostor may try to defeat our verification process by giving correct answers, especially at the beginning of their attack. However continuous re-verification of paths should allow us to catch the impostor eventually (see Section 6.4.3).

We believe that path verification is itself an important research area and is part of our future work. The main objective of this work, however, is to show that path-filtering can be coupled with a verification technique in an effective way and it imposes very loose constraints on the verification technique. Furthermore, since the path-filtering approach can remove the need to perform verification on every path, we believe that it will make the cryptography-based verification schemes more practical.

6.4.5 Parameter Setting

Different parameter settings can result in different performance trade-offs. In this section, we identify these trade-offs.

The time period T may be on the order of days, weeks or even longer depending on the desired level of security and the projected overhead. In general, a

longer time period means less frequent verification. So, while a long T can keep the overhead low, it may result in weaker protection. T is set to one week in our experiments.

A larger U_{min} may result in smaller path filters since fewer paths will be considered as base paths. This may lead to lower reachability to the servers, but it also provides stronger protection against temporary or misconfigured paths. One can therefore select an appropriate value based on the required level of reachability and the desired level of security. We choose a U_{min} of 10% in our study, i.e., if a path is used cumulatively for more than 10% of the time in a time interval, it will be considered a base path.

Using an EWMA of $U(p)$ allows a path to stay in the filter for a period of time even if it is not used by the peer. Suppose path p is now obsolete and its current $U(p)$ is 1, then we will keep this path for a period of $\lceil \log(U_{min}) / \log(1-\alpha) \rceil \times T$ before eliminating it from the filter. Although we would still verify this path with a certain probability during this interval, it is possible that, before we detect the path is no longer valid, a malicious attacker injects this path into the network. To minimize the risk of accepting such a spoofed route, we could use a larger α, a larger U_{min} or a smaller T. However, the trade-off is that we may prematurely remove paths that will soon return to use.

In general, the parameter settings are closely related to the verification mechanism. A highly manual verification mechanism will likely perform best with longer threshold values that result in fewer verification requirements, but this also means that the system will be slower to adapt to new paths. On the other hand, a highly automated low overhead verification mechanism could work best with low threshold values. Any optimal threshold settings will be highly dependent on the overhead and reliability of the verification mechanism, but our

primary objective is to demonstrate the path-filtering approach is feasible.

6.5 Evaluation

In this section, we again use the BGP updates collected at RRC00 from Feb. 24, 2001 to Feb. 24, 2002 to evaluate the adaptive path filter. We simulate a router that peers with one of the ISPs in Table 6.1 and compare its routes to the root/gTLD DNS servers before and after filtering.

The filter parameters are shown in Table 6.2. Note that these values are intended to demonstrate the feasibility of the path-filtering approach and do not suggest any optimal choice. In particular, the conservative values we chose were shown to work well on 12 months of actual data and the process triggered verifications that could have been handled in a non-automated, semi-automated, or fully-automated fashion.

T	T_r	U_{min}	α	P_v
1 week	1 hour	10%	0.25	0.1

Table 6.2: Parameter Setting

For this study, the monitoring process identifies and reports to the verification process any new paths that exist long enough to warrant further checking, but since we are using archived log data we cannot directly apply a verification process to the new paths. Therefore, we assume that the paths were accepted as valid. It remains part of the future work to evaluate the verification mechanism on peering sessions with operational BGP routers. However, the focus of this work is not the verification process, but rather the feasibility of the path filtering approach.

Our results show that the filter is able to reject many transient paths while

adapting to long-term path changes. For example, the simulated router filtered out most of the short-lived back-up paths to the A root server that were announced by ISP1 and automatically adapted to the three actual routing changes described in Section 6.2.2. As another example, over time ISP2 announced a total of 44 different paths to the A root server, seven of the 44 are stable paths. When our router peered with ISP2, it screened out 25 of the 44 paths and all the 7 stable paths were adopted as valid routes in the filter. In the remainder of this section, we first examine the nature of those paths that were filtered out and then present the filter's impact on root/gTLD server reachability.

6.5.1 Filtering Invalid Routes

In this section, we show that the paths rejected by our filter are mainly invalid paths such as those involved in Multiple Origin AS (MOAS) conflicts due to operation errors [ZPW01] or are routes that appear during slow BGP routing convergence.

6.5.1.1 Invalid Routes Due to Misconfiguration

On April 6,2001, an AS mistakenly originated a false path to DNS "C" gTLD server. Out of the 9 ASes we observed, 4 of them selected this bogus path as the best path to reach "C" gTLD server. However when the filter is applied to the BGP updates, all but one of the false announcements by the faulty AS were blocked out; that one skipped path lasted for more than 3 hours and was forwarded to the verification process.

Time	BGP Path
12:35:30	3549 19836 19836 19836 19836
16:06:32	3549 10913 10913 10913 10913 10913 10913 10913 19836
16:06:59	3549 1239 10913 19836
16:07:30	3549 701 10913 10913 19836
16:08:30	Path Withdrawn
16:15:55	3549 19836 19836 19836 19836

Figure 6.6: A Slow Convergence Example

6.5.1.2 BGP Slow Convergence

Figure 6.6 shows the BGP update sequence in a typical BGP slow convergence scenario (see Fig. 6.7 for the topology of the ASes involved the slow convergence). On June 19, 2001, AS3549 was using the path $(3549, 19836, 19836, 19836, 19836)$ to reach the DNS A root server. From this AS path, we can see that AS19836 originated this path and AS3549 directly peers with AS19836. At 16:06:32, AS3549 announced another path to the A root server, implying it could no longer reach the server through the direct peering with AS19836. This type of path change might be due to a break down in the link between AS3549 and AS19836, but BGP logs show that several ISPs that did not depend on the AS3549-AS19836 connectivity also lost their routes to the A root server at the same time. Thus it is most likely that the problem was inside AS19836. Nevertheless AS3549 continued in vain to explore all possible paths through its other neighbor ASes and generated additional closely spaced new path announcements. After trying all its back-up routes, AS3549 finally sent a path withdrawal message.

BGP log data showed that this incidence of slow convergence produced 24 unnecessary BGP updates from the nine peers and our filter was able to filter

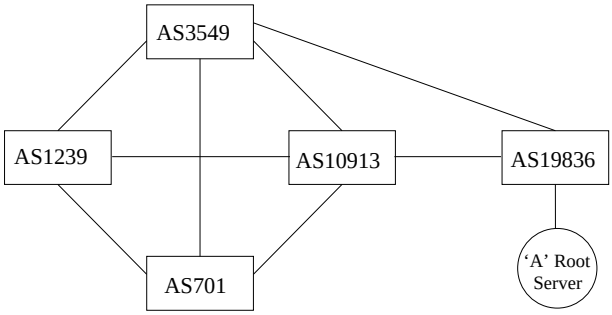


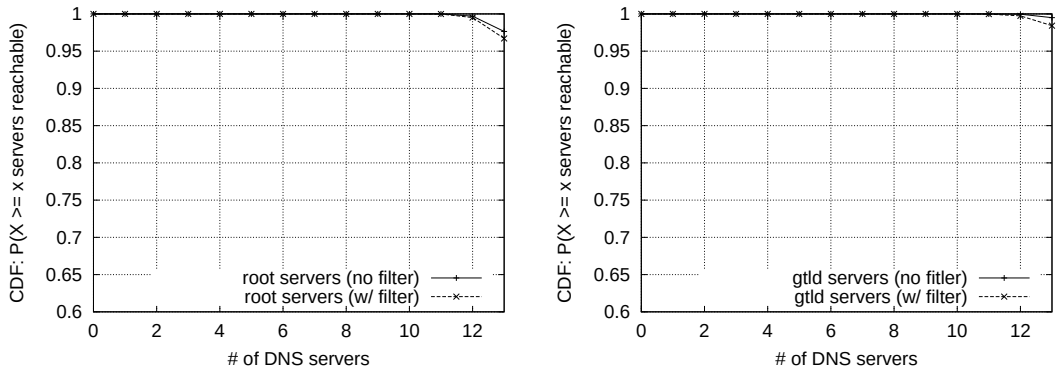
Figure 6.7: Slow Convergence Topology

out all of them. In other words, if deployed in the current Internet, the filter would have effectively blocked these false updates from further propagation, thus reducing the processing load at BGP routers and possibly improving the routing convergence time.

6.5.2 Impact on Server Reachability

In this section, we compare the difference in server reachability with and without using the filter. Ideally, the use of filter should cause negligible decrease in reachability. We first show the results for ISP1 and ISP2, then we summarize results for the other ISPs.

Fig. 6.8(a) shows the reachability to the DNS root servers through ISP1. The x-axis is the number of servers and y-axis is the percentage of time when x or more root/gTLD servers are reachable. The solid and the dashed curves correspond to the reachability before and after we applied the filter, respectively. The two curves overlap until x is 10. For x equal to 11, there is a very small difference: 99.997% without filtering and 99.982% with filtering. The difference becomes 0.948% for x equal to 13 root servers. It is evident that the filter has little impact on the router's reachability to the root servers. Since the graph for



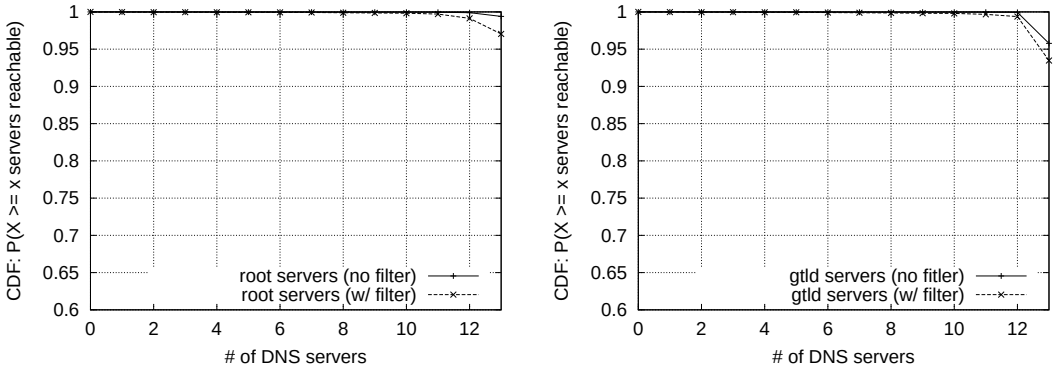
(a) (b)

Figure 6.8: Reachability through ISP1: (a) root servers; and (b) gTLD servers

the gTLD servers (Fig. 6.8(b)) is quite similar to that for the root servers, we do not discuss it in detail here.

The overall reachability to the root servers through ISP2 was not as good as through ISP1 during the examined period (see Fig. 6.9(a)). There were a total of 503 seconds during which no routes to any of the DNS servers were available. A closer examination of the BGP log data shows that for several times ISP2 withdrew the routes to *all* the DNS servers at about the same time and then announced routes again shortly after. It is not clear to us yet why ISP2 exhibited such behavior. After we applied the filter to BGP updates from ISP2, the reachability to at least one server through ISP2 decreased from 99.998% to 99.974% (a difference of 2.1 hours). The difference between the two curves remains below 0.766% (67.1 hours) for x up to 12, and it becomes 2.359% (206.6 hours) for x equal to 13. Note that the difference in time is accumulated over the entire 12-month period and that the DNS service is available as long as one of the root servers is reachable.

The results for all the nine ISPs are summarized in Table 6.3. We only present the reachability to the root servers here (the results for the gTLD servers



(a) (b)

Figure 6.9: Reachability through ISP2: (a) root servers; and (b) gTLD servers

are similar). We make the following observations:

1. One could reach at least one root server through six of the nine ISPs 100% of the time after we applied the filter. The same statistic for the other three ISPs is 99.974% for ISP2, 99.978% for ISP5 and 99.996% for ISP8.

2. One could reach at least 6 root servers through either ISP1 or ISP3 100% of the time even after we applied the filter, while the same statistic for the other ISPs ranges from 99.855% to 99.992%. This is because ISP1 and ISP3 (both are US-based global ISPs) had considerably better reachability to the root servers before filtering was applied (see Fig. 6.8).

3. The percentage of time to reach all the 13 root servers through ISP8 is very low (25.728%). This is mainly due to ISP8's poor reachability to the DNS "E" root server: it had no route to this server 71.5% of the time during our study period. The percentage of time when all the 13 servers were reachable through the other eight ISPs ranges from 90.228% to 97.780%.

It is essential that a network be able to reach at least one DNS root/gTLD

No. Servers	N=1		N=6		N=13	
	Unfiltered	Filtered	Unfiltered	Filtered	Unfiltered	Filtered
ISP1	100%	100%	100%	100%	97.649%	96.701%
ISP2	99.998%	99.974%	99.972%	99.939%	99.398%	97.039%
ISP3	100%	100%	100%	100%	95.149%	93.974%
ISP4	100%	100%	99.940%	99.855%	98.017%	95.981%
ISP5	100%	99.978%	99.951%	99.947%	91.341%	90.228%
ISP6	100%	100%	99.397%	99.397%	98.808%	97.248%
ISP7	100%	100%	99.998%	99.966%	99.535%	97.395%
ISP8	100%	99.996%	99.975%	99.971%	25.728%	25.419%
ISP9	100%	100%	99.994%	99.992%	99.475%	97.780%

Table 6.3: Percentage of Time when N or More DNS Root Servers are Reachable

server at all times in order for DNS to operate correctly. As we have seen, six of the nine ISPs we studied allow our router to satisfy this requirement after adding the filter protection. If a network has less than ideal reachability to the DNS servers even without the filter, it needs to improve its reachability first.

It is also important to keep in mind that we measured the reachability through a *single* ISP, but in reality an ISP peers with multiple ISPs and multi-homed client ASes are becoming a common case. Thus we expect the actual reachability to top level DNS servers to be much higher than the values we report here, both before and after the deployment of our path-filtering protection mechanism.

6.5.3 Impact on Route Adaptation Delay

This work takes advantage of the high redundancy level in DNS server implementations and suggests a heuristic approach to the adoption of all new route

changes. We insert a short delay before accepting any new routes, which damps out short-lived transient routes that may be caused by benign configuration errors, by protocol design or implementation errors that may lead to transient dynamics, or by malicious attacks. More importantly, this short delay in new route adoption offers an opportunity to verify the validity of the new change through some other channels.

Unfortunately when this temporary delay applies to each hop along the path of a routing change propagation, the total adaptation delay grows linearly with the path length. In our specific case of protecting DNS service, a new path to any top level DNS server(s) may not be used immediately. However, we argue that

1. Recent studies have shown that the Internet topology is becoming more richly connected. [MF02] shows that the average AS path is only 3.5 hops and 93.5% of the AS paths are below 6 hops (DNS server paths may have an even smaller average length as they are usually located in well-connected networks). Therefore, the routing adaptation delay caused by the path filter is expected to be short and bounded.

2. Due to high DNS server redundancy, this short delay in adapting to a new path should have a low impact on DNS service.

We believe that the resilience enhancement to faults and attacks by our path-filtering mechanism is far more important than a slightly faster adaptation to a new path. If one is willing to trade off the resilience for shorter adaptation delay, one possibility is to accept all new routing changes immediately and start the new route validation process at the same time.

6.6 Summary

We have presented an adaptive path-filtering mechanism to guard the DNS root and gTLD service against faults and attacks in network routing. Our design exploits the high degree of redundancy in the top level DNS system and the stability in network connectivity to the server locations. The proposed mechanism filters out potentially invalid routes by restricting the routes to the top-level DNS servers to change within a set of established routes. The set of established routes can adapt to long-term topology changes by adding only verified persistent routes. We have evaluated the feasibility and effectiveness of path-filtering mechanism using 12 months of BGP route logs. The results show that our design effectively detects the insertions of invalid routes with little impact on DNS service reachability. Furthermore, the path-filtering mechanism is simple and readily deployable by ISPs. Even after DNSSEC and other cryptography-based mechanisms become widely deployed, our approach will still provide an important added line of protection for DNS service.

CHAPTER 7

Fast Routing Table Recovery

In this chapter we present Fast Routing Table Recovery (FRTR), a scalable mechanism for detecting and correcting route inconsistencies between neighboring BGP routers. Rather than attempting to identify and eliminate *all* the potential faults and attacks that might cause inconsistencies, which is an impractical goal in the context of a large scale, distributed system such as the Internet, we take the approach that unexpected faults and attacks are inevitable, thus inconsistencies *will* occur. Given that the large size of BGP's global routing table makes the typical soft-state solution of periodic updates infeasible, FRTR uses Bloom filter [Blo70] to efficiently encode routing table data. BGP neighbors periodically exchange their Bloom-filter digests to detect any *potential* routing inconsistencies. After a session reset, FRTR uses digests to identify which routes have changed and sends only those routes. In addition, to overcome the false-positive drawback of Bloom filters, FRTR "salts" the digests with random seeds and periodically changes the seeds to ensure strong consistency between BGP routers.

This chapter is organized as follows. Section 7.1 introduces new terms and definitions. Section 7.2 describes the FRTR design. Section 7.3 provides more details on the protocol and implementation. Section 7.4 shows an example of how FRTR works. Section 7.5 describes how we used routing tables collected from ISPs to evaluate FRTR. Section 7.6 and 7.7 present the results. Section 7.8 summarizes the contributions of this work. This chapter is based on [WMP04].

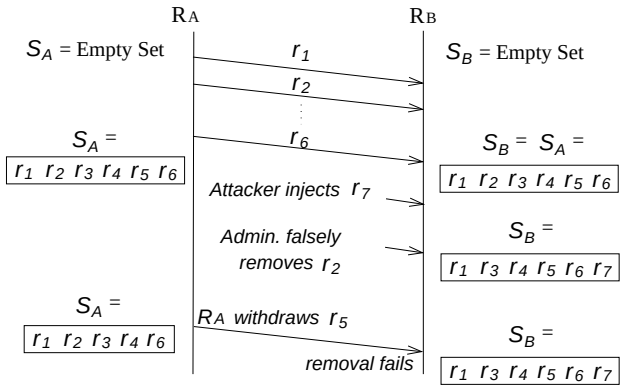


Figure 7.1: An Example of Routing Faults

7.1 Terms and Definitions

Neighboring BGP routers exchange routes and store them in Routing Information Bases (RIBs). If R_A and R_B are two neighboring routers, the set of routes that R_A sends to R_B is denoted $RibOut_{A,B}$. The set of routes that R_B learned from R_A is denoted $RibIn_{B,A}$. BGP supports *export* and *import routing policies*; an *export policy* controls which routes to send to each neighbor and an *import policy* decides which received routes to save and use. We make two assumptions: (1) R_A applies its export policy to a route before putting it in $RibOut_{A,B}$; and (2) R_B stores a received route in $RibIn_{B,A}$ before applying its import policy. Section 7.3.4 presents solutions when the above assumptions do not hold.

Ideally, $RibOut_{A,B} = RibIn_{B,A}$. However, faults or attacks may lead to inconsistencies between them. Examples of such faults and attacks include, but are not limited to, memory corruption, failure to remove a stale route, and insertion of an invalid route. Figure 7.1 illustrates how $RibIn_{B,A}$ may become inconsistent with $RibOut_{A,B}$. Since we focus on the communication between two routers, we use the simplified terms $RibOut_A$ and $RibIn_B$ in the remainder of this chapter.

Let S denote a set of n routes $\{r_1, r_2, \ldots, r_n\}$. Each route r_i consists of a

network address prefix ($Prefix(r_i)$) and a set of path attributes ($Attr(r_i)$). We define the following types of changes to S that can be caused by faults or attacks:

- *Insertion* of r_{n+1} into S: $S' = S \bigcup \{r_{n+1}\}$;
- *Modification* of r_i in S: $S' = S - \{r_i\} \bigcup \{r_i'\}$, where $Prefix(r_i') = Prefix(r_i)$ and $Attr(r_i') \neq Attr(r_i)$;
- *Removal* of r_i from S: $S' = S - \{r_i\}$.

7.2 FRTR Design

There are two existing approaches to achieving routing table consistency. The first approach is to let neighboring routers periodically send their complete routes or full connectivity information to each other. When the routing table size is large, however, this brute-force approach incurs a high cost. The second approach, which is adopted by OSPF[Moy98], uses a checksum to verify whether a piece of routing information is corrupted. In OSPF, a router computes a checksum over its link state and sends both of them to its neighbor. The neighbor then periodically computes a new checksum over the link state, and whenever the two checksums do not match, it discards the link state. This scheme detects unexpected internal changes, such as memory corruption, but offers no protection if a link state is accidentally removed along with its checksum, or if an obsolete link state fails to be removed.

Our proposal, FRTR, unifies the above two approaches and at the same time addresses their limitations. In FRTR, each router computes a digest over its routes using Bloom filter [Blo70]. Neighboring routers then exchange routing digests periodically to protect against unexpected insertion, removal, or modifi-

cation of their routing state. Bloom filter maps each route to only a few bits in the digest, making the periodic exchanges both effective and efficient.

In the following sections, we first describe how the digest mechanism works in one round of message exchange. We then show how periodic digest messages with changing "salt" values ensure consistency between neighboring routers. Finally, we show how routers can efficiently synchronize their routing tables after a session reset.

7.2.1 FRTR Digest Exchange Steps

Step 1: Computing the Sender Digest d_A

The sender, R_A, computes a digest d_A over $RibOut_A$ and sends the digest to neighbor R_B. Suppose R_A uses an l-bit digest (d) and k hash functions ($h_1, h_2, \ldots, h_k$) to encode a set of n routes (S). The digest is initially set to all zero. For each route $r \in S$, R_A first computes the k hash values $h_1(r), h_2(r), \ldots, h_k(r)$ and then sets the corresponding bits in the digest. In other words, if $d(i)$ designates the i^{th} bit in d,

$$d(h_i(r)) = 1, 1 \leq i \leq k.$$

For example, if one of r's hash values is 65, then the 65^{th} bit of the digest to 1. Note that the hash values of other routes may map to the bits that have already been set to 1, in which case those bits simply remain to be 1. Figure 7.2 illustrates how the digest is computed.

Step 2: Identifying Invalid Routes

The neighbor, R_B, receives d_A and uses it to determine whether its $RibIn_B$ contains any routes not currently used by R_A, i.e. the set difference ($RibIn_B -$

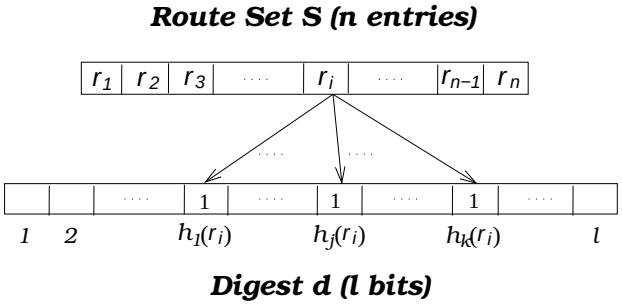


Figure 7.2: Digestion Computation

$RibOut_A$). More specifically, for a given route r in $RibIn_B$, R_B first computes the k hash values of r and checks whether the corresponding bits in the digest d_A are set to 1. It then places r into one of the following two groups:

1. **Invalid Routes**: $\exists i, 1 \leq i \leq k$, such that $d_A(h_i(r)) = 0$. Since Bloom filter does not produce any false negatives, R_B can be certain that $r \notin RibOut_A$;

2. **Probably Valid Routes**: $\forall i, 1 \leq i \leq k$, $d_A(h_i(r)) = 1$. It is probable that $r \in RibOut_A$, but r could also be a false positive.

R_B also computes its own digest d_B in the above process by setting the corresponding bits in d_B whenever it identifies a probably valid route.

When R_A's Bloom filter has a low false positive rate, R_B can have a high probability of identifying all the invalid routes in $RibIn_B$. The false positive rate of a Bloom filter is determined by the *encoding ratio* (l/n) and by the number of hash functions (k). It can be computed as follows. First, let p denote the probability that $d(i) = 0$ after the digest is computed.

$$p = (1 - \frac{1}{l})^{k \times n} \approx e^{-k \times \frac{n}{l}} \tag{7.1}$$

The false positive rate f is the probability P($\forall i$, $1 \leq i \leq k$, $d(h_i(r)) = 1$) given that r is not a member of the set in question, i.e.,

$$f = (1 - p)^k \approx (1 - e^{-k \times \frac{n}{l}})^k \tag{7.2}$$

Let's use α to denote the encoding ratio (i.e. l/n). The false positive rate f is minimized when $k = ln2 \cdot \alpha$ (see [BM02]). In other words, for a given α, there exists an optimal number of hash functions that minimizes the false positive rate. For example, when the encoding ratio is 5, the optimal k is 3.47 (in practice either 3 or 4 is used). Alternatively, one can fix the number of hash functions and adjust the encoding ratio to keep the false positive rate below a target value. For example, if the number of hash function is 3, one can use an encoding ratio of 8 to keep the false positive rate below 3%.

Step 3: Detecting Missing Routes

After removing the invalid routes in Step 2, R_B proceeds to determine whether any routes are missing, i.e. whether $RibOut_A - RibIn_B \neq \emptyset$. One way to test the above hypothesis is to see whether $RibOut_A$ and $RibIn_B$ have the same digest. Note that R_B has already computed its own digest d_B over all the probably valid routes in Step 2.

If $d_A \neq d_B$, we can be certain that $RibOut_A - RibIn_B \neq \emptyset$. However, if $d_A = d_B$, we cannot conclude that all routes from $RibOut_A$ are present in $RibIn_B$. The accuracy of this test depends on the false positive rate of the Bloom filter and the size of $RibOut_A - RibIn_B$; a lower false positive rate and a bigger difference between $RibOut_A$ and $RibIn_B$ both result in higher testing accuracy.

Step 4: Recovering Missing Routes

The goal of this step is to recover any missing routes. Let P_A denote the list of prefixes in $RibOut_A$ and P_B denote the list of prefixes in $RibIn_B$ (excluding the prefixes of invalid routes). If $d_B \neq d_A$, R_B sends P_B to R_A. R_A then checks every prefix $p \in P_A$ and classifies p as follows:

1. **Missing Prefix**: If $p \notin P_B$, R_B has no route to this prefix (or had an incorrect route that was removed in Step 2). R_A needs to re-advertise the route to prefix p.

2. **Probably Received Prefix**: If $p \in P_B$, R_B has a route to this prefix and the corresponding path attributes are likely to be correct (since otherwise the route would have likely failed step 2). Nothing needs to be done in this case.

In addition, if $p \in P_B$ but $p \notin P_A$, we have identified an invalid route in R_B. This can occur if the inconsistency was not detected in Step 2 due to a false positive. To remove this invalid route, R_A simply sends a BGP withdrawal message to R_B.

7.2.2 Periodic Updates

Since one cannot predict how or when a route may be corrupted, error detection and recovery must be done periodically. The FRTR design has R_A send periodic updates containing only d_A, the digest of $RibOut_A$, thus keeping the overhead low.

As with all other periodic refresh schemes, the interval between periodic updates represents an engineering tradeoff. Frequent updates allow routers to detect

faults quickly, but incur a higher bandwidth and processing overhead. On the other hand, infrequent periodic messages introduce less overhead, but the average time before an error is detected is increased. Nevertheless, FRTR with a long refresh period is still a qualitative improvement over the current BGP in which unexpected errors stay permanently until the next session reset. Note the trade-off is not necessarily a one-time fixed decision. For example, given a bandwidth budget, [SEF97] discusses how to adjust the soft-state rate based on the number of messages to send.

As we mentioned earlier, Bloom-filter digests can lead to false positives in both Step 2 and Step 3, especially when small size digest is used to keep the overhead low. FRTR periodically sends digests, but simply resending the same digest will result in the same false positives. In order to catch those false positives, FRTR design uses *"salted"* MD5 hash functions. MD5 [Riv92] was chosen because its computation is fast and several widely available hardware and software implementations exist. The salt is a randomly generated 32-bit value which is prepended to every route so that the MD5 computation will produce different hash values for the same route when the salt changes. The salt values can be either negotiated by the two neighboring routers beforehand or carried in every digest. Adding the salt enables new digests to be generated in each periodic exchange, which significantly reduces the chance that a false positive from one round would remain as a false positive in the next exchange.

The salted digests introduce another engineering tradeoff. By sending periodic updates with salted digests, a false positive in one round can be detected in subsequent rounds. Thus one can choose a smaller digest for reduced overhead and still achieve a good long-term error recovery rate. The cost is a longer delay before an inconsistency is detected.

7.2.3 Recovery after a Session Reset

After the peering session between R_A and R_B goes down, R_B marks all the routes in its $RibIn$ as obsolete[1]. It also starts a timer for the removal of these routes in case the peering session remains down for an extended period of time. When the timer expires, all the routes marked as obsolete will be flushed from the $RibIn$. The setting of this timer should be negotiated between the two routers so that R_B does not prematurely timeout the routes.

When the session comes up, R_A sends R_B its digest and R_B checks whether any routes have become invalid. If a route can be matched to the digest, its status will be changed from obsolete to valid. At the end of this process, R_B removes any routes still marked as obsolete. Now if R_B's digest still does not match R_A's, it sends a request to R_A for the missing routes.

The recovery process in FRTR is much more efficient than the full routing table exchange currently used by BGP. In FRTR, R_A sends only the digest and the routes that have indeed changed during the session down time. More importantly, BGP routing convergence will be much faster. Although there is a small probability that some stale routes may not be removed after the first exchange due to false positives, these routes will be removed in the subsequent exchanges. In addition, the periodic digests may be sent more frequently immediately following a session reset to ensure routing convergence.

[1]The BGP implementation needs to decide whether to use these routes for data forwarding during the recovery time, but this issue is orthogonal to our work.

7.3 Design and Implementation Specifics

7.3.1 Route Groups

If we compute a digest over a *RibOut* that has over 120K routes, the digest would exceed the BGP message size limit and must be sent in a series of fragments. This is generally considered undesirable because the receiver has to wait till all the individual pieces have arrived before it can start processing the digest.

A better approach is to divide the *RibOut* into multiple groups by the prefix ranges and then process the routes sequentially in each group, so that the digest for each group of routes can fit into one BGP message. When the sender transmits a digest to the receiver, it also includes in the message the starting and ending prefixes of the corresponding route group. The receiver sorts its routes in the same order. When it receives the digest message, it uses the starting and ending prefix to identify which routes in its $RibIn$ should be matched to the digest.

This optimization can significantly reduce the bandwidth overhead needed for error recovery. In a straight-forward implementation, in order to identify a single missing route, the receiver needs to send the prefix list of all the probably valid routes in its $RibIn$ to the sender. By dividing its routing table into multiple route groups, the receiver can associate the missing route with a specific route group and therefore much less information needs to be exchanged.

However, this optimization requires that the sender and receiver be able to sort their *RibOut* and $RibIn$. Explicit sorting is not needed if BGP implementations organize their routing tables using a Patricia trie structure, as in the routing software Zebra [Zeb], since an in-order walk of the tree will produce a list of routes sorted by the prefixes. For a Patricia trie structure $RibIn/RibOut$ containing N routes, the digest can be computed in time $O(N)$ as the tree walk visits every

route only once. Note also that even if the BGP implementation does not use Patricia trees and cannot sort the routes easily, it can still use FRTR without the optimization.

7.3.2 Incremental Digest Computation

In the basic design, we recompute the digests before they are sent because the salt value is changed in every round of checking. To reduce the computation overhead, one may choose to change the salt value less frequently, say every N rounds, and compute the digests incrementally before the salt changes. The tradeoff is longer time to detect a false positive.

To allow incremental digest computation, we use a counter for each bit in a digest (similar to [FCA00]). This counter records how many times the corresponding bit has been set to 1. When a new route is added, the bits corresponding to its hash values will be set to 1 and their counters increased by 1. When a route changes, the counters corresponding to the old/new hash values will be decreased/increased by 1. When a counter reaches 0, the corresponding bit will be set to 0.

7.3.3 Precomputed Digests

Although the previous optimization can reduce a router's digest computation overhead, its peer still needs to check every route in its $RibIn$ to detect route inconsistencies. To reduce its processing overhead in error detection and recovery, the peer can precompute its own digests (using the incremental computation method). When it receives a digest, it only needs to compare the received digest with its own to detect if any inconsistencies exist in the corresponding route group. And it needs to process individual routes only when the digests mismatch.

This optimization requires that the two neighbors agree on how their routing tables are divided into groups beforehand.

7.3.4 Policy Related Issues

A BGP router can have multiple peers and, according to its export routing policies, it may send different sets of routes to different peers. Network operators usually configure BGP peers by routing policies, grouping together peers with the same export policy so that they can share one *RibOut*. In this case, the router only needs to compute one set of digests per peer group.

A peer router may discard some of the received routes according to its import policy. If it computes a digest over only the saved routes, this digest will not match the sender's. One solution is to use *Cooperative Route Filtering* [CR03] so that the sender sends only those routes that match the receiver's import policy.

In addition, the peer router may modify some of the received routes according to its import policy, e.g. attach a community attribute to a route. Such modification will also lead to digest mismatch. One solution is to turn on the "Soft Reconfiguration Inbound" option provided in most BGP implementations; this option lets a router save a copy of all the pre-policy routes.

We propose a solution that uses a new BGP path attribute and eliminates the need to save the pre-policy routes. In this approach, the sender first uses a hash function to compute a *path-attribute hash* that covers only the path attributes associated with the route. This hash function is independent of the hash functions used in the digest computation. The hash value is then stored in a new BGP attribute called *attr_hash* and transmitted along with the route announcement. The FRTR digest is then computed using the 3-tuple $\langle salt, prefix, attr\_hash \rangle$. The receiver must store a copy of *attr_hash* and must not modify it. How-

ever, any other path attributes can be modified or discarded by the receiver. When checking the digest, the receiver computes its digest using its copy of the $\langle salt, prefix, attr\_hash \rangle$.

To ensure full protection against all types of corruptions, the receiver needs to verify the $attr\_hash$ value when the route is first received. Before modifying or discarding any path attributes, the receiver should compute its own hash over the unaltered path attributes and verify that this hash matches the value in $attr\_hash$. The path attributes can then be modified or discarded. Subsequent digest checks only require the $attr\_hash$ value.

7.3.5 BGP Messages and Message Order

FRTR defines two new BGP message types: *Digest* and *Prefix*. They both have a common BGP header. A Digest message contains a digest, as well as the hash functions and salt value used in the digest computation if they are not pre-negotiated. A Prefix message contains a list of prefixes whose routes match the peer's digest. Both message types also contain two prefixes to specify the corresponding group of routes.

Due to the minimum delay imposed on the propagation of route changes, a router may send a Digest message before previous route changes have been propagated and the peer may unnecessarily invoke the recovery process to synchronize its routes. BGP implementations should be designed to avoid this kind of problem. More specifically, the Digest message should be sent only after all the existing changes to the corresponding route group have been propagated to the peer(s). In addition, new changes to the route group should not be allowed while the digest is being computed and those route changes should be sent only after the Digest message is sent.

7.4 An Example of FRTR Usage

From the RIPE RCC00 monitoring point [RIP], we obtained a routing table containing 101,404 routes. The routing table was used by a router (R_A) in an operational ISP. We now use it to illustrate how FRTR works assuming that the optimization described in Section 7.3.1 is implemented.

Suppose we use a digest size of 1024 bytes and an encoding ratio of 5. These parameters allow each digest to encode $1024*8/5 = 1638$ routes and the router R_A can encode its entire table in 62 digests. In each round of consistency checking, R_A sends 62 Digest messages to its peer(s). The generation and transmission of these messages may be paced to avoid congestion.

The first Digest message from R_A covers the routes from 4.0.0.0/8 to 24.240.125.0/24. When R_A's peer R_B receives this message, it first locates the starting prefix 4.0.0.0/8 in the corresponding $RibIn$. Then it computes the hash values of every route between 4.0.0.0/8 and 24.240.125.0/24. If any of the routes does not match the digest, that route is removed. For example, if an attacker injected a forged route for the prefix 4.0.0.0/8, this route most likely will not match the digest and therefore will be removed.

To detect missing routes, the peer R_B computes its own digest in the above process. If the result does not match R_A's, R_B sends a Prefix message to request for missing routes. The Prefix message contains all the prefixes whose routes match R_A's digest. For example, if the route to prefix 6.1.0.0/16 was mistakenly removed by an administrator, the Prefix message will not contain 6.1.0.0/16. R_A will notice this prefix is missing by comparing the received Prefix message with the list of prefixes in its $RibOut$.

The last Digest message from R_A carries a flag that indicates the end of the

routing table. After processing this message, R_B removes any routes that have not been matched to any of the received digests.

7.5 Data Source and Methodology

This study also uses routing data from the RRC00 monitoring point at RIPE NCC [RIP]. However, instead of the logged routing updates, we use RRC00's *routing table* recorded at 16:00 GMT on Jan. 20, 2003. Table 7.1 shows the eleven peers that were active at that time. To obtain the routing table of a monitored router, we simply group the routes in RRC00's routing table according to the advertiser's IP address. The number of routes in the eleven derived routing tables ranges from 101,404 to 119,750.

Location	AS Number (ISP)
US	AS7018 (AT&T), AS2914 (Verio), AS3549 (Glocal Crossing)
Netherlands	AS3333 (RIPE NCC), AS1103 (SURFnet)
Switzerland	AS513 (CERN), AS9177 (Nextra)
Britain	AS3549 (Global Crossing)
Germany	AS13129 (Global Access)
Japan	AS4777 (NSPIXP2)
Australia	AS4608 (APNIC)

Table 7.1: RRC00's Active Peers (Jan. 20, 2003)

We use a script to emulate two peering routers R_A and R_B. R_A adopts one of the routing tables obtained from the RRC00 monitoring point and advertises all the routes to R_B. In the first part of our evaluation (Section 7.6), we assume the BGP session between R_A and R_B fails with a given rate, and compare the

bandwidth overhead of FRTR with that of the current BGP. In the second part of our evaluation (Section 7.7), we introduce random errors into R_B's $RibIn$ and measure the error recovery ratio and bandwidth overhead of FRTR. We assign a probability of error P_e and an error type to the routing table, i.e. there is a probability of P_e for generating an error of the given type for each route in the routing table.

Four types of errors are used in our experiments: *removal, insertion, modification and mixed errors*. An error generated for a route r has the following effects on r depending on the error type:

- **Removal**: remove r from the routing table;
- **Insertion**: insert a more specific route of r into the routing table. For example, if r's prefix is 129.250.0.0/16, a route to the prefix 129.250.0.0/17 will be inserted;
- **Modification**: modify r's path attributes;
- **Mixed Errors**: first randomly choose one of the above three types of errors with equal probability, then introduce the chosen error to the routing table.

We simulate random errors here because they are easier to generate and analyze. In real networks, errors may be correlated. For example, the error probability for route A may be positively correlated with that for route B. However, we do not expect the results to be significantly different because the hash functions we use can generate highly random hash values. In other words, even though route A and B are likely to have errors at the same time, their probability of being a false positive is not correlated.

7.5.1 Parameter Setting

We choose the digest size (l) to be 1,024 bytes so that each Digest message is well within the size limit of BGP messages – 4096 bytes. We use two encoding ratios (α): 5 and 8. Therefore , a digest can encode 1638 routes ($\alpha = 5$) or 1024 routes ($\alpha = 8$). These encoding ratios are not meant to be the optimal values, but are used to illustrate the tradeoff between the various performance metrics. To produce a digest for a group of routes, we first calculate the MD5 signature of each route and then take three 13-bit values (i.e. $k = 3$) from the MD5 signature as the hash values.

7.5.2 Performance Metrics

We compare the *bandwidth overhead* of FRTR with that of a full table exchange. In FRTR, the Digest and Prefix messages are overhead as they do not directly correct errors. In a full BGP table exchange, any BGP routing update that does not correct an error is considered bandwidth overhead. We also measure the *error recovery ratio* of FRTR. More specifically, we calculate the percentage of errors that are corrected after each round of digest exchange.

7.6 Senario 1: BGP Session with Transient Failures

In this section, we consider how well FRTR works in recovering from events such as congestion and link failures that cause a peering session to fail. Routing table inconsistencies caused by other types of faults such as memory corruption and attacks will be considered in the next section.

We estimate the long-term bandwidth overhead of FRTR given a session failure rate of λ and compare it with that of the current BGP. In FRTR, the band-

width overhead is the total size of the Digest and Prefix messages. Let's denote them B_d and B_p. We ignore B_p in the following analysis since, in the case of transient session failures, B_p is usually much smaller than B_d . If the Digest messages are periodically sent at a rate β, then the bandwidth overhead of running FRTR is $B_d \cdot (\lambda + \beta)$.

In the current BGP, R_A needs to send its routing table to R_B after their session fails. Let's denote the size of this table exchange B_t. Suppose the fraction of routes that actually need to be updated is q, then the overhead is $B_t \cdot (1-q) \cdot \lambda$.

We are now interested in the ratio between the two bandwidth costs, i.e. $\frac{B_d \cdot (\lambda+\beta)}{B_t \cdot (1-q) \cdot \lambda}$. Let's use AS2914 as an example. The total size of its FRTR digest messages is $65,151$ bytes when the encoding ratio is 5. On the other hand, advertising its entire table would consume $4,980,127$ bytes of bandwidth. Therefore, $B_d/B_t = 0.013$. To estimate q, we need to know how many prefixes have route changes over a short period of time, e.g. 3 minutes. According to RISReport [RIS], AS2914 sent BGP announcements to 69 unique prefixes per minute on average from Mar 7, 2004 to Apr. 7, 2004. Since this number includes duplicate announcements, it overestimates the number of actual route changes. In other words, we can expect 207 or fewer prefixes to have route changes (0.2% of AS2914's routing table), if the session to AS2914 is recovered within three minute. Therefore, q is very close to 0 and the ratio becomes $0.013 \cdot (1 + \beta/\lambda)$. We make two observations from this result:

1. When β is 0, FRTR consumes only 1.3% of the bandwidth overhead of the current BGP in AS2914's case. In other words, if used only after session resets, FRTR cuts down the overhead of routing table synchronization by a factor of 77.

2. When β is non-zero, the overhead of FRTR depends on both the session

failure rate and the frequency of periodic Digest messages. If the session fails once a day, FRTR will achieve a lower overhead if the digests are sent once every 19 minutes (or less frequently). Note that a session failure rate of once a day is not uncommon. Some links in operational networks have a failure rate much higher than that (see [ICM02]). Moreover, regular maintenance and policy changes can also lead to session resets.

7.7 Scenario 2: BGP Table Corruption

We now consider the performance of FRTR in recovering corrupted routing tables when a variety of errors are introduced. The performance results for one round of recovery are presented in Section 7.7.1 and 7.7.2, and the results for multiple rounds are presented in Section 7.7.3. Since we obtained similar results for all the eleven routing tables, we present only the results for AS2914's routing table here for brevity.

7.7.1 Error Recovery Ratio

Figures 7.3–7.6 show the percentage of errors corrected using FRTR. The X-axis is the probability of error (P_e) in log scale. We have chosen 9 different P_e's in the range of [0.0001, 0.9]. For each P_e, we perform 30 simulation runs to obtain the 95% confidence interval of the mean error recovery ratio. The two curves in each figure correspond to the encoding ratio of 5 and 8 respectively.

7.7.1.1 Removal Errors

Figure 7.3 shows the recovery ratio for removal errors. When P_e is 0.0001, the recovery ratio is around 92.4% ($\alpha = 5$) and 96.9% ($\alpha = 8$). Both curves increase

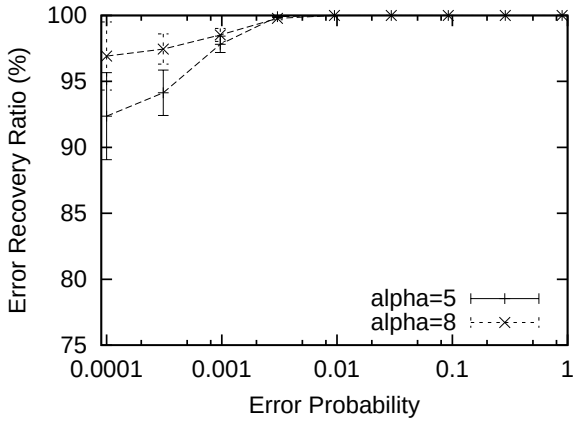


Figure 7.3: Recovery Ratio of Removal Errors

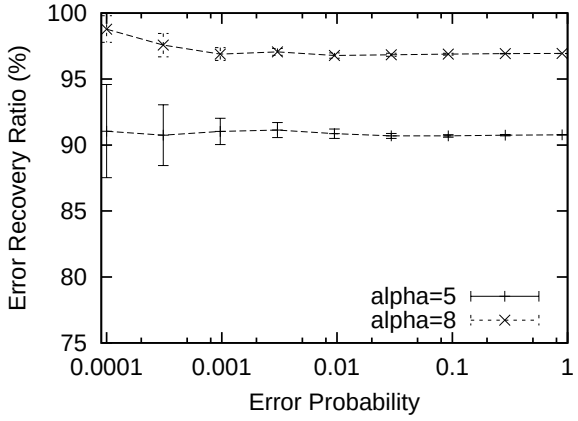


Figure 7.4: Recovery Ratio of Insertion Errors

to 100% when P_e reaches 0.003, and stay at 100% for higher error probabilities. This is because, with only removal errors, all the errors are detected through the "missing routes test", i.e. Step 3 of the digest exchange process (see Section 7.2.1). As P_e increases, more routes are removed from R_B's routing table. This larger difference leads to a higher accuracy in the test.

7.7.1.2 Insertion Errors

The insertion errors show very different characteristics: the error recovery ratio stays around 91% ($\alpha = 5$) and 97% ($\alpha = 8$) regardless of the error probability (see Figure 7.4). This is because we evaluate a different step of the digest exchange process here. In this experiment, there are no missing routes so the "missing routes test" is irrelevant. Instead, the inserted routes are detected by checking their hash values against the digest from R_A (i.e. Step 2 of the digest exchange process). The lower the false positive rate with regard to R_A's digest, the higher the percentage of inserted routes detected using this type of checking.

We can compute the theoretical expected false positive rate using Equation 7.2. When α is 5, $f = (1 - e^{-k \times \frac{n}{l}})^k = (1 - e^{-k/\alpha})^k = (1 - e^{-3/5})^3 = 0.0918$. The error recovery ratio should be equal to $1 - f \approx 91\%$. When α is 8, the false positive rate f is 0.03 and the error recovery ratio should be roughly 97%. Both numbers match our experimental results. Furthermore, since the false positive rate depends not on the error probability, but on the parameters used in the digest computation (i.e. α and k), the error recovery ratio does not change with the error probability.

7.7.1.3 Modification Errors

Similar to those curves in Figure 7.3, the two curves in Figure 7.5 have an increase at the beginning. And similar to those curves in Figure 7.4, they stay around a particular value afterwards (91% for $\alpha = 5$ and 97% for $\alpha = 8$). This is because both Step 2 and 3 are tested in this experiment and the error recovery ratio is affected by the failure rate of both steps.

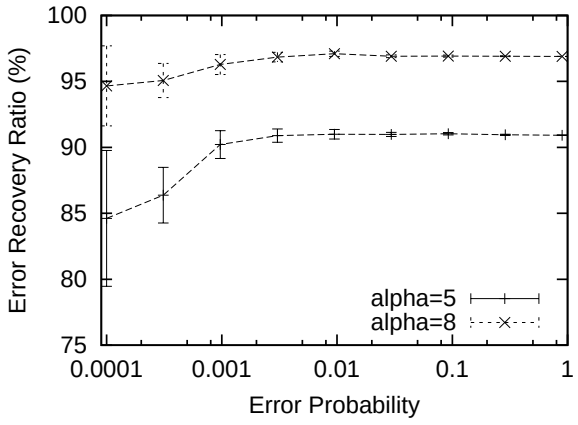


Figure 7.5: Recovery Ratio of Modification Errors

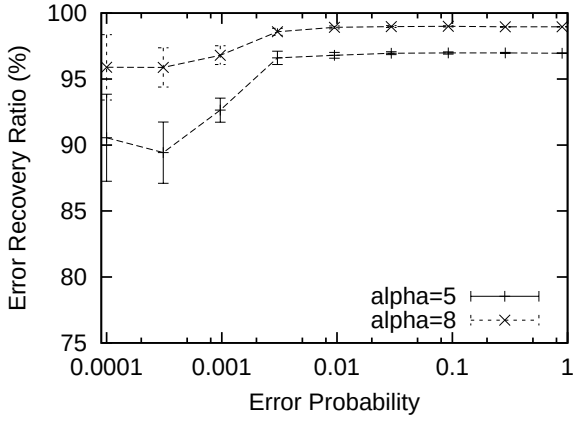


Figure 7.6: Recovery Ratio of Mixed Errors

7.7.1.4 Mixed Errors

Figure 7.6 shows that when α is 5, the curve increases from around 90% to 97% and stays there, and when α is 8, the curve increases from around 96% to 99% and stays there. We can expect that, if we have a different combination of errors, the curves may move up or down depending on which type of errors is dominant. This is because the result is roughly a combination of the error recovery ratios of the different types of errors.

7.7.1.5 Summary

The error recovery ratio depends on the specific type or combination of errors, as well as the parameters in the digest computation. With a low encoding ratio of five and only three hash functions, we can correct at least 91% of the errors most of the time and achieve a 100% error recovery ratio for removal errors when the error probability is higher than 0.003. Furthermore, increasing the encoding ratio to 8 can significantly increase the error recovery ratio to be around or higher than 97% for most error types and error probabilities.

7.7.2 Bandwidth Overhead

Figure 7.7 shows that FRTR with an encoding ratio of 5 has a much lower overhead than a full BGP table exchange for most error probabilities. We explain the difference between the two in greater detail below.

First, *insertion errors* incur the lowest bandwidth overhead (Figure 7.7(b) shows the widest gap between the curves). This is because only Digest messages were sent in FRTR to correct the insertion errors, i.e. no Prefix messages were triggered (see Section 7.7.1). The Digest messages consume a constant 65,151 bytes of bandwidth which is only 1.3% of the bandwidth overhead required by the table exchange (4,980,127 bytes).

Secondly, for removal errors, the maximum bandwidth overhead of FRTR (460,609 bytes) is reached when the error probability is around 0.009. However, it is still only 9% of the bandwidth overhead of a full table exchange under the same error probability. As the error probability approaches 0.9, the gap between the two curves gets smaller as one would expect, but the full table exchange still has a higher overhead. Modification and mixed errors show similar characteristics

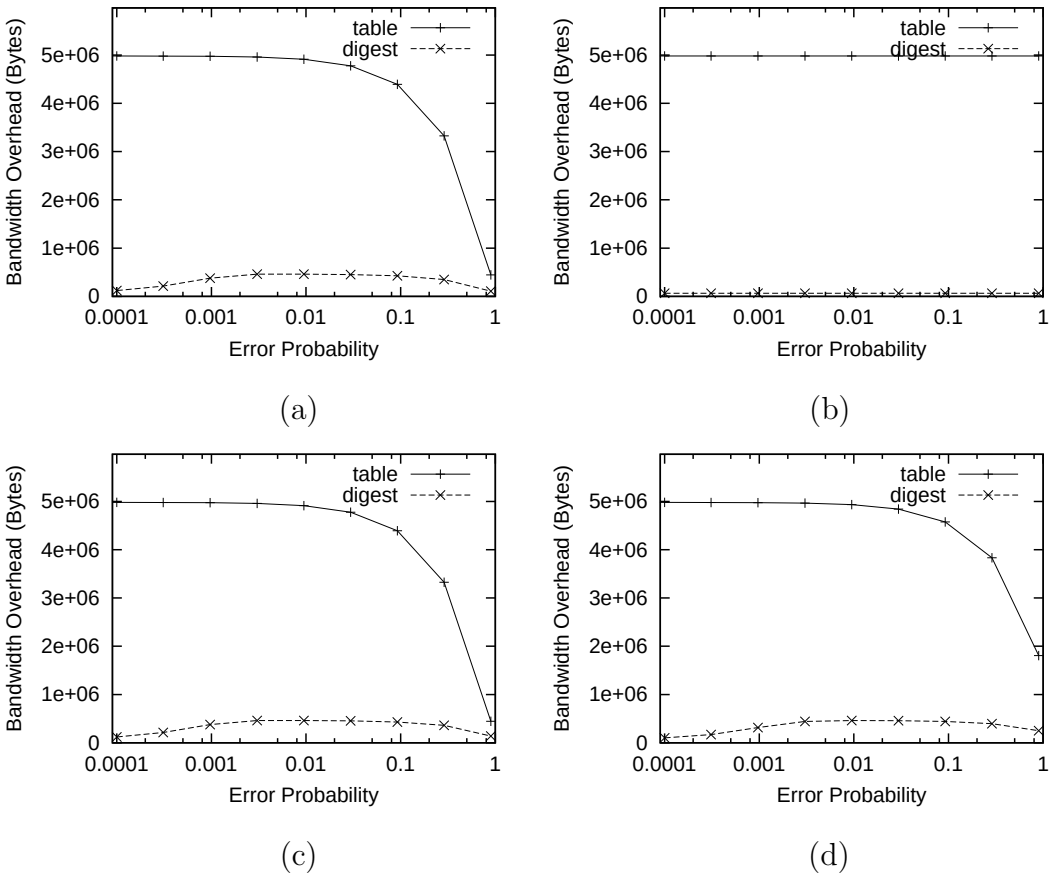


Figure 7.7: Bandwidth Overhead ($\alpha = 5$): (a) removal errors; (b) insertion errors; (c) modification errors; and (d) mixed errors.

as removal errors.

The bandwidth overhead of FRTR with an encoding ratio of 8 is still much lower than that of a full BGP table exchange (see Figure 7.8). The higher encoding ratio increased the bandwidth overhead by less than 42K bytes for all error types and error probabilities.

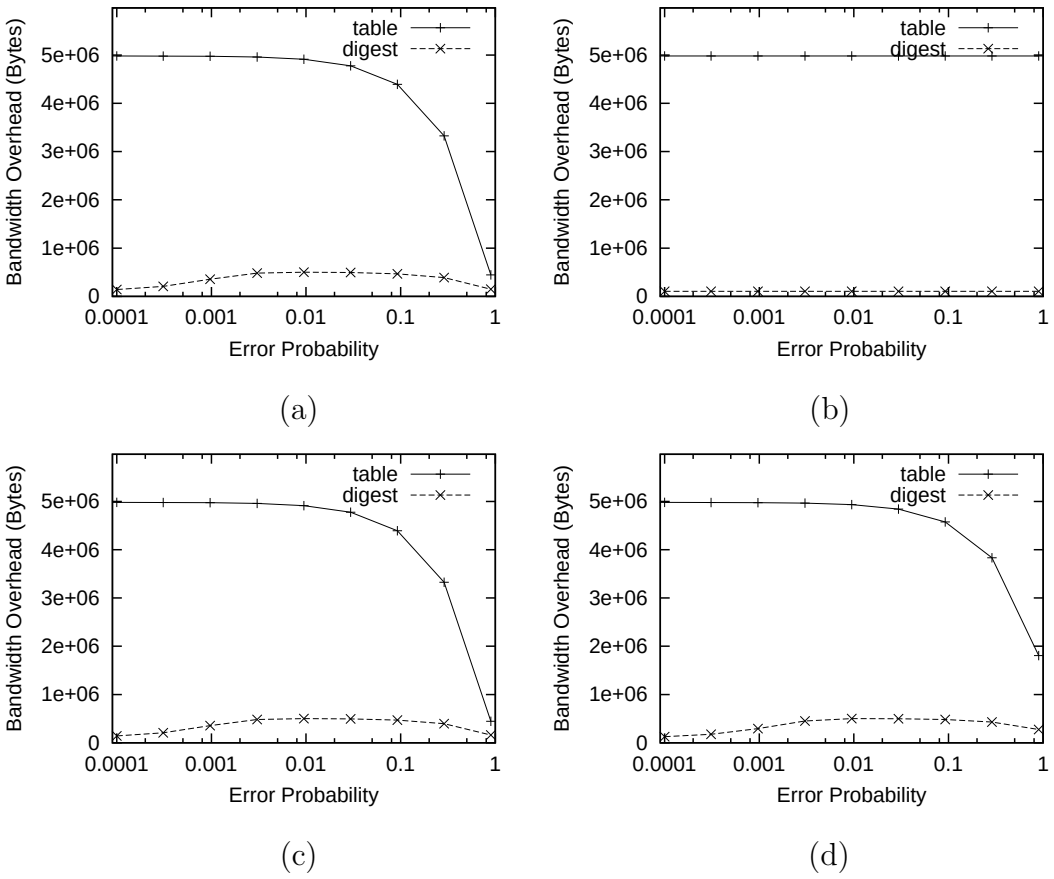


Figure 7.8: Bandwidth Overhead ($\alpha = 8$): (a) removal errors; (b) insertion errors; (c) modification errors; and (d) mixed errors.

7.7.3 Multiple Rounds of Recovery

In the previous experiments, we have demonstrated that FRTR can achieve a high error recovery ratio with a low bandwidth overhead after one round of error detection and recovery. However, it is still necessary to let neighboring routers periodically exchange the Digest messages to correct any new errors that have crept in since the last digest exchange, as well as to correct any errors that were left undetected previously, such as those due to Bloom filter's false positive errors.

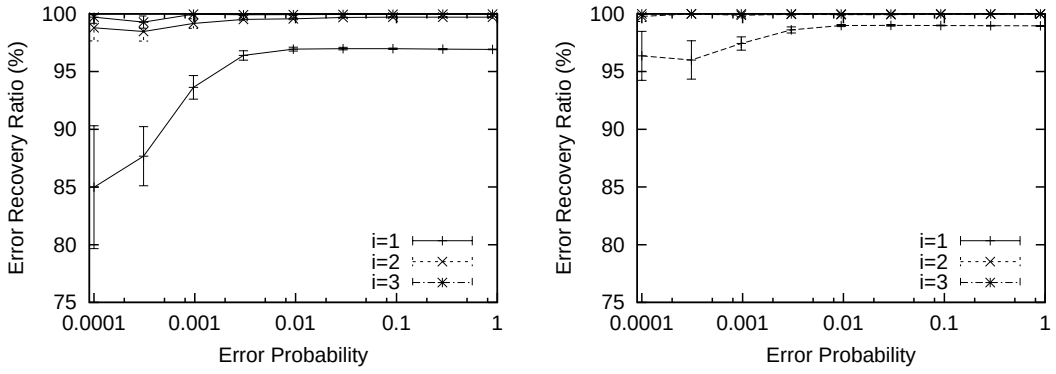


Figure 7.9: Recovery Ratio of Mixed Errors after Multiple Rounds: (a) $\alpha = 5$; and (b) $\alpha = 8$.

To evaluate the performance of the periodic "salted" digests, we run FRTR for three consecutive rounds with a range of error probabilities and measure the percentage of corrected errors after each round. For easier understanding of the results, we do not introduce new errors in the second and third round.

In Figure 7.9(a) and (b), we show the error recovery ratio of mixed errors when α is 5 and 8 respectively. The results for other error types are similar, so we omit them for brevity. The curves labeled "$i = 1$", "$i = 2$" and "$i = 3$" correspond to the results after the first, second and third round. The two figures show that, when the encoding ratio is 5 (or 8), we can correct more than 99.9% (or 99.99%) of the errors in most cases after three rounds of error detection and recovery. This matches our analytical result — if the hash values generated in one round are independent from those in the next round, the percentage of uncorrected errors after i rounds should be $(1 - g)^i$ where g is the one-round error recovery ratio (g is approximately 91% and 97% for α of 5 and 8 respectively).

We also observe that the advantage of the higher encoding ratio diminishes quickly with multiple rounds of recovery. A practical implication of this result is

that one can use a Bloom filter with a low bandwidth overhead to achieve good long-term performance.

7.8 Summary

FRTR is a *fast* and *bandwidth efficient* mechanism to detect inconsistencies between neighboring routers and resynchronize their routing tables whenever inconsistencies are detected. FRTR encodes routing table state using Bloom filter to efficiently detect and recover otherwise unnoticeable errors. Furthermore, FRTR takes advantage of periodic exchanges by salting the digests to effectively eliminate false positives. Finally, FRTR can significantly reduce the overhead of routing table recovery after BGP session failures.

We have evaluated FRTR design through both analysis and simulation. Our results show that, with one round of digest exchange, FRTR can detect and recover more than 91% of random errors and the overhead can be as low as 1.3% of a full routing table exchange; a slight increase in the digest size can achieve a detection and recovery rate higher than 97%. Furthermore, the use of salted digests allows FRTR to ensure nearly 100% consistency between neighboring routers after only a few rounds of digest exchange. By comparison, the current BGP would not detect any of these errors. Finally, FRTR facilitates incremental deployment since any two *neighboring* routers can start using FRTR when they both implement the scheme.

Previously Fan et al. applied Bloom filters to a collaborative web caching system to reduce coordination overhead [FCA00]. In this system, each web cache keeps a Bloom filter of the URLs of its cached webpages and, instead of sending a list of the URLs, it sends its Bloom filter to neighboring caches. A cache

can check the Bloom filters received from its neighbors to find out whether a requested page is available at one of its neighbors. Our work also makes use of the space-efficiency of Bloom filters. However, our problem is more complex than simple membership lookups — given a corrupted data set (the routing table), our challenge is to identify all the routes that are not members of a given set and to find out the routes that are missing from the given set.

Our work is closely related to [BCM02] that proposes several data structures including Bloom filter for approximate reconciliation of set differences. There are three key differences between FRTR and [BCM02]. First, since BGP routing tables change dynamically in both the number of routes and the attributes of each routes, our solution must be able to accommodate such dynamics changes easily. The hierarchical structures proposed in [BCM02] are inappropriate for FRTR because any changes in a route will move the route to another part in the data structure. Second, the simplicity of using a flat Bloom filter is preferred for implementation in large scale routers where more complex data structures and algorithms tend to cause more errors in implementation and operations. Third, since FRTR addresses a problem in an existing network protocol, it faces design issues not considered in [BCM02]. For example, to prevent the reassembly of large digest messages, FRTR divides the routing table into groups and computes one digest over each group to make each digest fit into one message. Furthermore, we also note that, contrary to the statement in [BCM02] that Bloom filter works well only when there is a large number of differences, our work shows that Bloom filter's performance is adequate even when the error probability is only 0.0001 (about 10 errors for a BGP table with 100K routes).

Although our design has been presented in the context of ensuring routing state consistency *between* BGP neighbors, the same techniques we developed in

this work can also be used *within* a router to ensure the consistency between its internal routing table and forwarding table. Furthermore, the basic approach developed in FRTR should be applicable to other protocols where a strong state consistency among multiple entities must be enforced.

CHAPTER 8

Scalable RSVP Refreshes

The RSVP protocol shares the same challenge as BGP in keeping a large amount of state synchronized between neighboring routers. RSVP was designed with the traditional periodic update approach, which leads to a high overhead as the number of reservation sessions goes up. One may reduce the overhead by using a longer refresh period, which unfortunately leads to longer delays in re-synchronizing RSVP state.

To overcome the dilemma between protocol overhead and responsiveness, we apply "state-compression" to RSVP overhead reduction. Our "Scalable RSVP Refresh" mechanism is similar to FRTR in that it replaces all the refresh messages sent between two neighboring nodes for each of the RSVP sessions with a *digest* message that contains a compressed "snap shot" of all the shared RSVP sessions between two neighbor nodes. When an RSVP node, say N, receives a digest from a neighbor node, it compares the value carried in the digest message with the value computed from N's local RSVP state. If the two values agree, N refreshes all the corresponding local state; otherwise N starts a state re-synchronization process to discover and repair the inconsistency. To assure quick state synchronization in face of packet losses we also enhance RSVP with an acknowledgment option, so that instead of waiting for next refresh, any lost RSVP messages can be quickly retransmitted. Our mechanisms achieve a constant message transmission overhead and low delay while retaining the resilience of the original RSVP

protocol.

This chapter is structured as follows: we briefly introduce the basic RSVP terminology below. In Section 8.2 we present an overview of our algorithm. Section 8.3 describes in detail the data structure and procedure for computing RSVP digests. Section 8.4 presents a list of proposed new RSVP messages and related processing rules that are needed to implement our new scheme. Section 8.5 analyzes the overhead of digest computation. We identify the limitations of our scheme in Section 8.6. Finally, Section 8.7 summarizes the contributions of this work. This chapter is based on our work in [WTZ99].

8.1 Terms and Definitions

RSVP is a receiver-driven protocol. To provide receiver-driven reservation functionality, a data source sends PATH messages towards the receivers, leaving behind a trace of "path state" at each router they traversed. Receivers wishing to make a reservation then send RESV messages, which follow the path state traces upstream towards the data source, reserving resources at each intermediate node along the way. The state set up by PATH and RESV messages is called *path* and *reservation* state, respectively. The state is deleted if no matching refresh messages arrive before the expiration of its life timer. The state may also be deleted by either the sender or receiver using PathTear or ResvTear messages.

PATH and RESV messages are idempotent. When a route changes, the next PATH message will initialize the path state on the new route, and future RESV messages will establish reservation state there; the state on the now-unused segment of the route will time out. Thus, whether a message is "new" or a "refresh" is determined separately at each node, depending upon the existence of state at

that node.

Before we proceed with our proposed RSVP state compression algorithm we give the definitions of some terms that will be used in the rest of this paper.

RSVP State An RSVP path or reservation state.

Regular/Raw RSVP Message RSVP messages defined in RFC2205 [BZB97], e.g. PATH, RESV, PathTear and ResvTear messages.

Refresh Message An RSVP message triggered by a refresh timeout to refresh one or a set of RSVP state. It can be a PATH message for a path state, a RESV message for a reservation state or a Digest message (in our scheme) for aggregate state.

MD5 Signature The result of the computation of the MD5 algorithm.

Digest A set of MD5 signatures that represents a compressed version of the RSVP state shared between two neighboring RSVP nodes.

8.2 Design Overview

The goal of our proposal is to improve RSVP's scalability allowing efficient operation with large number of sessions (e.g. tens of thousands sessions). More specifically, we aim at reducing the number of refresh messages while still preserving the soft-state paradigm of RSVP. In this section we briefly describe our *state compression* approach; the details of the compression scheme are presented in the next section.

Instead of sending a refresh message per sender-session pair to a neighbor, our approach is to let each RSVP node send a *digest* message which is a compact

way of representing all the RSVP session state shared between two neighboring nodes. In this way, the number of refresh messages per refresh period is reduced from being proportional to the number of sessions to being proportional to the number of neighbor nodes. Raw RSVP messages are sent either when triggered by state changes or after state inconsistency is detected to re-synchronize the state shared between two nodes.

These benefits cannot come without any overhead. Generally speaking, the protocol overhead of RSVP can be divided into two components, the *bandwidth overhead* for message transmissions, and the *computation overhead* for processing these messages. One can further subdivide the computation overhead to system overhead (e.g. system interrupts by packet arrivals) and message processing overhead. The state compression scheme can effectively decrease the bandwidth and system overhead, however at the cost of increased message processing overhead as we apply additional processing to compress RSVP state to a single digest message per neighbor. Therefore, one important part of our design is to minimize the cost of digest computation.

To compress RSVP state into a digest, one can simply concatenate the state of all the RSVP sessions into a long byte stream and compute a digest over it. However this brute-force approach suffers from a high overhead of recomputing the whole digest again whenever any change happens. To scale the digest computation we compute the digest in a structured way. First, we hash all the RSVP sessions into a table of fixed size. We then compute the signature of each session, and for each slot in the hash table we further compute the slot signature from the signatures of all the sessions hashed to that slot. On top of this set of signatures, we build an N-ary tree to compute the final digest (a complete description of the data structures used is given in section 8.3.3).

There are two advantages in using a tree structure to compute the digest:

1. Whenever the digests computed at two neighboring nodes differ, the two nodes can efficiently locate the portion of inconsistent state by walking down the digest tree;

2. When an RSVP session state is added/deleted/modified, an RSVP node only needs to update the signatures along one specific branch of the digest tree, i.e. the branch with the leaf node where the changed session resides.

In our current design, we use the MD5 algorithm to compute state signatures. As stated in [Riv92], "it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest." We can therefore conclude, with a high level of assurance, that no two sets of different RSVP states will result in the same signature. However, it should be noted that our state compression scheme can work well with any hash function that has a low collision probability, such as CRC-32, as long as two neighboring nodes agree upon their choice of the hash function.

As a further optimization, we also add an acknowledgment option (ACK) to the RSVP protocol. The ACK is used to minimize the re-synchronization delay after an explicit state change request. A node can request an ACK for each RSVP message that carries state-change information, and promptly retransmit the message until an acknowledgment is received. It is important to note the difference between a soft-state protocol with ACKs and a hard-state protocol. A hard-state protocol relies solely on reliable message transmission to *assume* synchronized state between entities. A soft-state protocol, on the other hand, uses ACKs simply to assure quick delivery of messages; it relies on periodic

refreshes to correct any potential state inconsistency that may occur even when messages are reliably delivered, for example state inconsistency due to undetected bit errors, or due to undetected state changes.

8.3 State Organization

One can suspect that the increase in refresh efficiency cannot come for free. This is indeed the truth and the trade-off comes in the form of increased storage and computation. The increase in storage originates from the need to keep *per neighbor state*, since separate digests are sent to different neighbors. Consequently, computation costs are inflated since we have to compute the per-neighbor digests and we have to operate on the per-neighbor data structures. In the sections that follow we elaborate on the requirements for extra state introduced by the compression mechanism. Computation costs are further analyzed in Section 8.5.

8.3.1 Neighbor Data Structure

Current RSVP implementations structure the RSVP state inside a node as a common pool of sessions, regardless of their destinations. On the other hand, digest messages sent towards a particular neighbor contain a compressed version of the RSVP state shared *with that neighbor*. The need therefore arises to further organize RSVP state inside a node according to the neighbor(s) each session comes from or goes to. To satisfy this need we introduce the *Neighbor* data structure which holds all the information needed to calculate, send and receive digests to and from a specific node.

In essence the Neighbor data structure is the collection of RSVP sessions that the current node sends to or receives from a particular neighbor. For efficiency

neighbor data structures may not actually store the sessions but contain pointers to the common pool of sessions. This way a session shared with multiple neighbors is not copied multiple times to the corresponding neighbor structures. In addition to sessions, the neighbor structure contains the digest computed from the sessions shared with the neighbor and some auxiliary information such as retransmission and cleanup timers.

A node needs to compute two digests for each neighbor, one for the state refreshed by messages received from that neighbor and one for the state the local node is responsible for refreshing towards that neighbor. We call these two digests InDigest and OutDigest respectively. OutDigest is sent in lieu of raw refreshes while InDigest is used to for comparison when receiving a Digest message from that neighbor. In the next section we present how we compress each session state into an MD5 signature. In section 8.3.3 we delve into the details of the data structure and algorithm we use to derive a digest from the session signatures.

8.3.2 Session Signature

To compress a session state into a signature, we first need to identify which session parameters need to be constantly synchronized between neighbors. Table 8.1 shows the RSVP objects included in the digest computation. A session is uniquely identified by a session object which contains the IP destination address, protocol ID and optionally a destination port number of the associated data packets. A Path State Block (PSB) is comprised of a sender template (i.e. IP address and port number of the sender), and a Tspec that describes the sender's traffic characteristics and possibly objects for policy control and advertisements. A Reservation State Block (RSB) contains filterspecs (i.e. sender templates) of the senders for which the reservation is intended, the reservation style and a

flowspec that quantifies the reservation. It may also contain objects for policy control and confirmation. Although PSBs and RSBs contain some other fields such as incoming interface and outgoing interfaces, these fields have only local meaning to a specific node and therefore should be excluded from the digest computation. As to RSVP objects defined in the future, the digest computation can also be applied to them if necessary.

RSVP Objects	Sub-objects to Include
Session	session object
PSB	sender template, sender tspec, adspec, policy
RSB	filterspec, flowspec, reservation style, policy

Table 8.1: RSVP Objects Included in Digest Computation

We noticed that, in the current RSVP specification, RSBs record only reservations requests received from downstream neighbors, but not reservation requests forwarded upstream. However, for a multicast session or many-to-one unicast session, the reservation request a node receives from a downstream neighbor may not be the same as the one it sends to an upstream neighbor if the node is a merging or splitting point. Since the sender of a digest has to compute the digest based on what flowspec and filterspec are sent to its neighbor, we require such information to be kept in associated RSBs to facilitate the digest computation.

8.3.3 Hash Table and Digest Tree

The existence of the structures described in this section is not fundamental for the correct operation of our compression scheme. However given the context where our proposed solution will be most useful (e.g. tens of thousands of sessions), these structures provide the desired performance to make the scheme practically

viable. Two are the principal reasons that impelled us to include these data structures:

- Given the need for expeditious response to state changes and the high volatility resulting from the high volume of sessions, updates, insertions and deletions must be done efficiently. This requirement can be translated to two subgoals: a data structure that supports efficient session insertions and deletions and second, *incremental* digest computation. Unfortunately, the design of the MD5 algorithm does not allow incremental digest computation. To overcome this limitation we compute the state digest *recursively*, by applying the algorithm to session sets of increasing size.

- State inconsistencies must be resolved rapidly without requiring complete state retransmission. To do so, we need to quickly locate which part of RSVP state contains the inconsistency and then send raw refreshes only for these sessions.

In addition to the two primary reasons, simplicity and robustness are essential if this mechanism is to supersede the minimalism and potency of raw refreshes. With this set of goals in mind, we continue by presenting each one of the two data structures next.

Sessions are stored inside a hash table. The size of the hash table is M and sessions are *hashed* to one of the M hash table slots. Hashing is done over some fixed session fields (e.g the session's address). If multiple sessions hash to the same slot, they are inserted in a linked list. Sessions inside the linked list are stored ordered according to their destination address. Figure 8.1 shows the session hash table. Slot i contains a pointer to the head of the linked list of all stored sessions that hash to i. It also contains an MD5 signature that is computed by

concatenating all the sessions' MD5 signatures and applying the MD5 algorithm on the compound message.

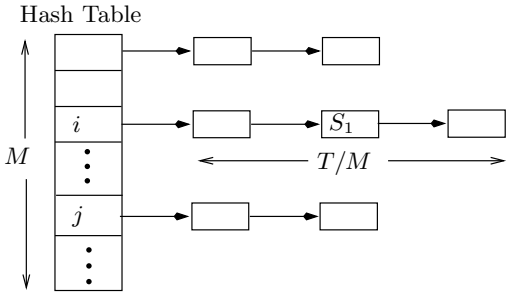


Figure 8.1: Hash Table

The second step is to reduce the total number of signatures from M to N, the number of signatures that can fit inside a single message. To do that we have introduced a complete N-ary tree whose leaves are the slots of the hash table. This *digest tree* is shown in Figure 8.2.

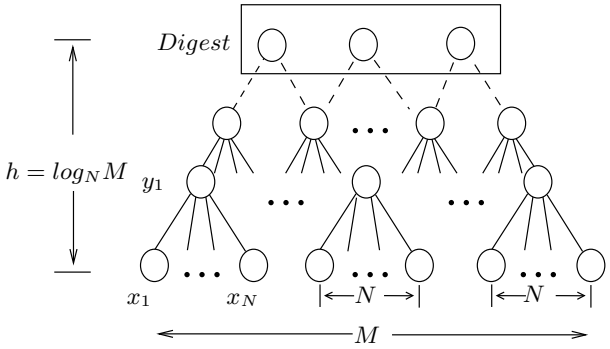


Figure 8.2: Digest Tree

A node constructs the digest tree in the following way. As we said earlier, the leaves of the tree are the signatures stored in the slots of the hash table. The signatures of N slots are concatenated and the MD5 algorithm is applied on the compound message. The result is stored at the parent node on the tree. Looking

at Figure 8.2, signatures $x_1, \ldots, x_N$ are concatenated and the MD5 algorithm result is stored in node y_1. This grouping results in M/N level-1 signatures. If the number of level-1 signatures is still larger than N, the node continues on to group each of N level-1 signatures to compute a level-2 signature to get M/N^2 level-2 signatures. If C_i is the number of level-i signatures, we repeat the grouping until C_i is less than or to equal to N. The top level signatures represent the *digest* of that RSVP state.

We have chosen the degree of the tree to be the same as the maximum number of MD5 signatures inside the digest object to simplify the data structure and to reduce the number of parameters. Note that all insertions and deletions are done in the hash table while the purpose of the digest tree is to reduce the number of signatures from M to N and to store intermediate results that will be used during the recovery phase, after inconsistencies are detected.

The hash table size M and the maximum number of signatures N in a digest are two important factors that affect the performance of our digest scheme. A smaller M results in more sessions being hashed to each slot on average, which means more overhead in updating the signature of a slot. However, given an N, a smaller M also leads to a lower digest tree and consequently fewer intermediate levels of the tree to maintain and fewer messages to exchange during the recovery procedure. This suggests that, when choosing M, one needs to take into consideration both the target tree height and the expected number of sessions in each hash slot. Furthermore, if the actual number of sessions differs greatly from the expected number of sessions, the sender of a digest may need to change its M to achieve better performance. As a result, the sender should nofify the receiver of the modified M and the receiver needs to use the new value of M in its digest computation. A larger N means each node in the tree has a larger fan-out and

therefore the digest tree will have fewer levels. In general, one would like N to be the largest value allowed by the link MTU.

8.4 Mechanism Description

8.4.1 New RSVP Messages and Objects

Our compression mechanism requires three new RSVP messages, namely: *Digest*, *ACK* and *DigestErr*. A Digest message carries a timestamp object that uniquely identifies it and a digest object that represents the state shared between a node and its neighbor (i.e. the receiver of the message). After a node discovers a neighbor capable of exchanging digests (see section 8.4.2), it periodically sends Digest messages refreshing the total RSVP state of that neighbor. If a node wishes to send Digest messages at a different interval than the standard, it can specify that interval in the Digest message. In this way, the receiver will know when to expect Digest messages and in their absence when to delete the associated state.

ACK messages are used to acknowledge raw RSVP messages or Digest messages. Since many messages may be outstanding when an ACK is received at the sender side, the ACK message contains the timestamp of the message it acknowledges. The receipt of an ACK message indicates that the original message was received and processed by the receiver. Moreover, the message was processed at the receiver side without creating any errors. Otherwise, an error message (ResvErr, PathErr or DigestErr) would have arrived instead of the ACK message.

A DigestErr message acts as a negative acknowledgment to a Digest message. Similar to ACK messages, the DigestErr message carries the timestamp of the received digest message. In addition, it contains the digest value computed at

the receiver side, which is used later in the recovery process (see section 8.4.4).

The timestamp object mentioned before contains two basic fields: the *Timestamp* field which is the time that the packet was sent and the *Epoch* field which is a random 32-bit value initialized at boot time. All timestamp objects sent from a node should use the same Epoch value as long as the node is not rebooted. If, after the initialization phase, a node receives two consecutive messages with different Epoch values, it can conclude that the sender of these messages has rebooted. The receiving node must then purge all state associated with that sender.

We chose to use time as the message identifier because it is always increasing and so a sender does not have to check if the value is in use or has been used before. It also helps the receiver to identify which of the RSVP messages for the same state is the most recent one. However, depending on the node's processing speed and timer granularity, two consecutive messages may get the same timestamp value. Therefore, we define the timestamp to be $max(t, t_{last} + 1)$, where t is the current time and t_{last} is the last timestamp value used.

Furthermore, the timestamp object carries a flag indicating whether the sender is requesting an acknowledgment for this message. This flag should be turned off in the timestamp objects carried by ACK and DigestErr messages to avoid an infinite exchange of ACK messages.

Last, the digest object carries a set of MD5 signatures. These signatures can be either the digest or some set of MD-5 signatures from some other level of the digest tree.

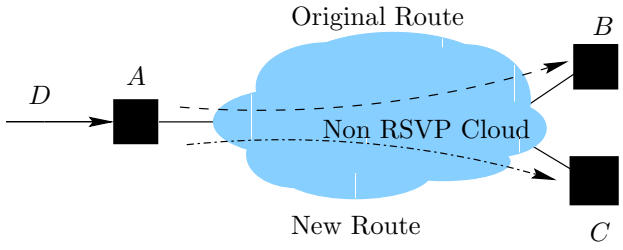


Figure 8.3: RSVP Session over a non-RSVP cloud

8.4.2 Neighbor Discovery

To use the compressed refresh scheme, a node needs to discover which of its neighbors are capable of exchanging digests. For this reason, when a RSVP node starts sending (raw) RSVP messages for a session, it should request that the neighbor(s) acknowledge these messages by including a timestamp object with the ACK_Requested flag turned on. If the node receives an ACK message in response from a neighbor whose address is not currently on the Neighbor Structure list, it has then discovered a new compression-capable neighbor. If on the other hand, that neighbor does not understand timestamp objects (legacy node), it will return an error message. We can then conclude that this neighbor is compression-incapable.

When a non-RSVP cloud exists between two RSVP neighbor nodes, although the nodes can discover each other using acknowledgments during the initial message exchanges, the upstream neighbor may not be able to detect whether sessions crossing the cloud switch next hops. These changes are caused by route changes inside the non-RSVP cloud and are not detectable if the upstream neighbor's outgoing interface remains the same. The original RSVP specification does not share this problem since RSVP messages for individual sessions carry the session's address and therefore naturally follow any route changes. In the compression

scheme however, digest messages are explicitly addressed to particular next hops and therefore the same solution cannot be used.

Figure 8.3 illustrates our point. In this scenario node A originally has B as its downstream neighbor for session D. After a route change, node C becomes A's downstream neighbor for that session. However, since A's outgoing interface remains unchanged, A will not notice the route change, hence it will continue to include session D when calculating the digest to B. Node B will not be informed of the change either as long as A sends it the same digest. Therefore, node C will never get a PATH message from A. As a result, resources will be reserved on the path between A and B while data packets will follow the path from A to C.

The digest scheme therefore, cannot be used over non-RSVP clouds until an effective way of detecting route changes is found. Fortunately, the existence of non-RSVP clouds can be detected by mechanisms described in [BZ97]. If a non-RSVP cloud exists between two nodes, regular refreshes should be used instead of the compression mechanism.

8.4.3 Normal Operation

Neighboring nodes start by exchanging regular RSVP messages as usual. Once a node discovers a compression-capable neighbor, it creates a digest for the part of its RSVP state that it shares with each of this neighbors. Subsequently, the node sends Digest messages instead of raw RSVP refreshes at regular refresh intervals. When an event that changes the RSVP state (e.g. a sender changes its traffic characteristics (Tspec)) occurs, raw RSVP messages are sent immediately to propagate this change.

Raw RSVP messages are sent as before, with the added option of asking for an ACK. A sender requesting an acknowledgment, includes in the message a

timestamp object with the ACK_Requested flag turned on. The sender also sets a retransmission timer for the packet sent. Processing at the receiver side includes updating the digest of the session that the message belongs to as well as updating the digest tree. If during processing a condition occurs that requires sending back an error message back to the sender (e.g a ResvErr) then the receiver sends back to the sender that error message. This error message will cancel any pending retransmissions of the original message.

If no ACK is received before the retransmission timer expires, the sender retransmits the message up to a configured number of times. Each of the retransmissions carries the same timestamp contained in the original message. If an updated message (i.e. a PATH message from the same sender but with different Tspec) is sent before an ACK is received, the original message becomes obsolete and no longer needs to be retransmitted. If no ACK arrives even after the message has been retransmitted for the maximum number of times, the message is purged from the node's list of pending messages. Any inconsistencies created by the possible loss of this message will be later resolved by digests.

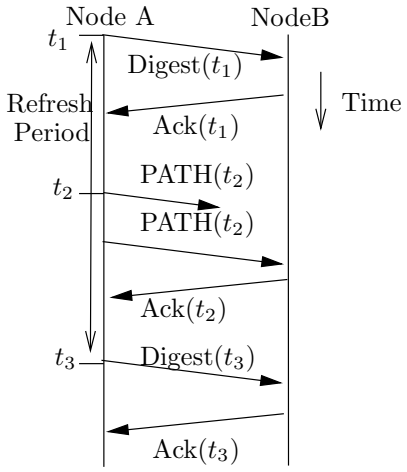


Figure 8.4: Message Exchange

Digest messages are always sent with the ACK_Requested flag turned on. Digest messages are also retransmitted for a maximum number of times in the absence of ACK messages. However, following the original RSVP design where an RSVP node never stops sending refresh messages for each active session, a node should not stop sending digest refreshes even if it fails to receive an acknowledgment in the previous refresh interval. If the neighbor node crashed and becomes alive again, it will find the digest value different from its own and the two routers will start the re-synchronization process. When the digest value is changed, the node needs to cancel any pending retransmission of the obsolete Digest message and promptly send a Digest message with the new digest value.

When a node receives a Digest message, it checks to see if the state reported by the Digest message is consistent with the corresponding state stored locally. To do so the node does a binary comparison between each of the MD5 signatures contained in the digest object and the corresponding MD5 signatures in the InDigest (see section 8.4.2). If all of them agree then the state is consistent and an ACK is sent back. Otherwise the receiver returns a DigestErr message containing its InDigest and the process described in the next section begins.

Figure 8.4 gives an example of message exchanges between two nodes under normal condition. Nodes A and B had consistent state at time t_1. A sent a Digest message to B and received an ACK message for it. A then had a state change at time t_2 which triggered a PATH message sent to B. This message was lost, so A didn't receive an ACK until it timed out and it had to retransmit the PATH message (using the same timestamp t_2). B received the retransmitted PATH message and sent an ACK message back to A. Up to this point, the two nodes were synchronized. When the digest refresh timer timed out at t_3, A sent a Digest message with updated digest value to B. Since A and B were still consistent, B

sent an ACK to A for the Digest message.

8.4.4 Recovery Operation

Two RSVP neighbors may become out-of-sync due to a number of reasons. For instance, a state-changing RSVP message got lost and the sender did not ask for ACK. It may also happen that a node crashed and lost part or all of its state. Since it is impossible to enumerate all the possible reasons, the best that one can do is to detect state inconsistencies once they arise and have a way of repairing the damaged state.

As we mentioned in section 8.4.3, a node sends a DigestErr message if the received digest value disagrees with the local digest. The timestamp and digest value in the DigestErr message help the two neighbors localize the problem. If the timestamp acknowledged is smaller than the timestamp of the last Digest message sent, this error message is for an obsolete message. This message should be ignored since it may not represent the current state of the neighbor. If they are equal, the node starts a depth-first search of the mismatching signatures from the root of the digest tree.

When a node receives a DigestErr message it compares the digest value with its own to find the states that are inconsistent. When it finds the first mismatching signature (call it S_1), it sends a Digest message containing the signatures used to compute S_1. A DigestErr is expected for this Digest message since at least one of the children signatures should not match. The node again looks for the first mismatching signature (S_2) in the second DigestErr message and sends the children of S_2 in a Digest message. This procedure is repeated until the leaf signature (S_h)[1] causing the problem is found. Now, the node knows that one or more of

[1] $h = \lceil log_N M \rceil$, see Figure 8.2

the sessions in that hash table slot (represented by S_h) must be inconsistent with those in the neighbor. It can then locate these sessions by further exchanging the session signatures with the receiver. However, we found that locating specific sessions may get quite complicated in some cases, for example, when the sender or receiver has sessions that do not exist in the peer. When a node encounters these cases, it can simply send raw refreshes for all the sessions in that particular bin. After refreshing these sessions, the node re-examines S_{h-1} (the parent of S_h) for other inconsistencies and continues to traverse the tree until all the mismatching sessions are located and refreshed.

Notice there is a tradeoff between the latency of the recovery procedure and the transmission efficiency. For example, if the tree has many levels, many RTTs are needed to exchange the digests at all the tree levels in order to find the leaf-level sessions that contribute to the inconsistency. However, if speed of convergence is more important than efficiency, one can stop at an intermediate tree level and refresh all the states represented by the mismatching signature at that level.

8.4.5 Time Parameters

There are two time parameters associated with digest messages: the refresh period between successive digest refreshes R and the retransmission timeout T. A node sends digests at intervals of r, where r is randomly chosen from the range $[0.5 * R, 1.5*R]$. Randomization is used to avoid the synchronization of digest messages. If an acknowledgment is not received after time T from the transmission of a digest, the node will retransmit that digest message.

The current RSVP specification [BZB97] states that the default refresh period for regular RSVP messages is 30 seconds but the interval "should be configurable

per interface". To be consistent, digest refreshes are also sent every 30 seconds by default and this interval should be configurable. As we mentioned in Section 8.4.3, digest messages are explicitly acknowledged and therefore there is no need to decrease R to protect against lost digest messages. However, R affects the frequency of consistency checking between neighbors, so smaller values of R should be used in environments where prolonged periods of inconsistency are undesirable. The retransmission timeout T should be proportional to the round-trip time between two directly connected neighbors. A node can measure the time interval between a message and the corresponding ACK and estimate the mean RTT by performing exponential averaging on the measurements.

Another important time parameter is the state lifetime L. If state represented by a digest is not refreshed for a period L, it is considered stale and is deleted. The naive approach would be to set L to be equal to the refresh period R. This would however lead to premature state time-outs at the receiving side. There are at least two reasons for this: first, clocks at neighboring nodes may drift and second as we said before the refresh timer is randomly set to a value in the range $[0.5 * R, 1.5 * R]$, which means that the sender may send digests at intervals larger than R. These examples illustrate that L should be larger than R. Following the current RSVP specification we decided to set $L = (K + 0.5) * 1.5 * R$, where $K = 3$.

8.4.6 Backward Compatibility

The extensions we have introduced are fully compatible with the existing version of RSVP. If an RSVP node sends a message with a timestamp object and subsequently receives an "Unknown Object Class" error, it should stop sending any more messages with attached timestamp object and start using regular refreshes

instead of digest refreshes. Digest messages do not pose a compatibility problem since a node will start sending Digest messages only when it discovers that its particular peer is compression-capable using the procedure outlined in section 8.4.2.

8.5 Computation Costs

In this section we focus on the operations applied to the data structures described in Section 8.3 and analyze their requirements in terms of processing.

We begin with some definitions. Let the number of sessions be T, the size of the hash table be M and the maximum number of MD5 signatures inside the digest message be N. Let's further define the cost of computing the MD5 signature of a message of size x to be $f(x)$. To determine the behavior of $f(x)$, we have to study the algorithm's behavior. Summarizing the description in RFC 1321 [Riv92], the algorithm divides the input message to 64-byte blocks and applies a sixty-four step process to each one of these 64-byte blocks. In each of the sixty-four steps, a number of bit-wise logical operations are applied to that 64-byte block. The results of the computation on the nth block are used as input for the computation of the $(n+1)$th block. After all the blocks have been processed, the message's signature is produced. From this description, one can see that $f(x)$ is a linear function of x, the size of the input message measured in bytes.

When a session is modified, a new signature for that session as well as a new digest of the whole RSVP state has to be computed. To illustrate this procedure, imagine that we want to update session S_1 inside the hash table of Figure 8.1. First, we look up the session inside the hash table. In our example, we would

come up with the index i. If multiple sessions map to the same hash table slot, we traverse the linked list of sessions until we find the session in question. Once the session is found and its new MD5 signature is computed, we have to compute the new MD5 signature stored at the base of the linked list which represents all the sessions mapped to that hash table slot. On the average $\lceil T/M \rceil$ sessions will occupy the same slot. The total time needed for this operation is therefore $f(16 * \lceil T/M \rceil)$, since each MD5 signature is 16 bytes long. The next step is to update the values on the digest tree. We begin by computing the MD5 signature of the contents of slot i concatenated with its $N-1$ *siblings* which will be stored in their *parent* node on the digest tree. We continue this procedure until we reach the top of the tree. Since there are $\lceil log_N(M) \rceil$ levels on the tree and at each level we apply the MD5 algorithm on a message of size $16 * N$ (the combined size of N MD5 signatures), the time spent during this step is $(\lceil log_N(M) \rceil - 1) * f(N * 16)$. Notice that the term is $\lceil log_N(M) \rceil - 1$ since we do not calculate an MD5 signatures out of the N topmost signatures.

From the discussion above, we can conclude that the total time needed to calculate the new digest after a session is modified is given by the following formula, where S is the size of a session in bytes:

$$f(S) + f(16 * \lceil T/M \rceil) + (\lceil log_N(M) \rceil - 1) * f(N * 16) \tag{8.1}$$

When a new session has to be inserted in the hash table, we locate the slot this session hashes to and insert the session to that slot's linked list, if one exists. Given that the list is ordered, the new session has to be inserted in order inside the list, which means traversing the list until we find a session whose destination address is larger than the destination address of the session we want to add and inserting the new session before that session. Deleting a session, involves finding the slot it hashes to, searching for it inside the linked list, and "splicing" its

predecessor to its successor on the list.

The computation cost for the creation of the new digest after an insertion or deletion operation, is almost identical to the update cost. The only difference is that in the case of deletion we don't calculate the MD5 signature of the session (since we are deleting it). Equations 8.2 and 8.3 respectively, show the insertion and deletion costs.

$$f(S) + f(16 * \lceil T/M \rceil) + (\lceil log_N(M) \rceil - 1) * f(N * 16) \tag{8.2}$$

$$f(16 * \lceil T/M \rceil) + (\lceil log_N(M) \rceil - 1) * f(N * 16) \tag{8.3}$$

We can see from Equations 8.1, 8.2 and 8.3 that when the size M of the hash table is small compared to the number of sessions T, the cost of updating the linked list of sessions will be linear to T. In this case, updating the linked list becomes the most expensive operation, forcing the total cost to also be linear to T. The size M of the hash table should therefore be comparable to T to avoid increased update complexities.

8.6 Limitations of Our Approach

The ability of two RSVP neighbors to exchange digests in place of raw RSVP messages relies on the assumption that the two nodes know precisely all the RSVP sessions that go through these two nodes in sequence. Whenever a route change occurs, the upstream node must be able to receive a notification from the RSVP/Routing interface and synchronize the state with the new downstream node (as well as tear down the session with the old downstream neighbor). For multicast sessions, another complication arises if a router is attached to a broadcast LAN. A router must detect all changes of membership in the downstream neighbors, for example when a downstream router on the broadcast LAN joins

or leaves a group, which does not affect the list of outgoing interfaces of the associated RSVP state. Again the proposed scheme relies on the RSVP/Routing interface to provide notification of such changes.

Furthermore, we have identified two cases where an RSVP node must resort to the current refresh scheme. The first case is when both compression-capable and compression-incapable downstream neighbors exist on the LAN. To accommodate the compression-incapable neighbors one must use per session RSVP refresh messages. The second case is when two RSVP nodes are interconnected through a non-RSVP cloud as we explained in Section 8.4.2.

In summary, a seemingly inevitable limitation of the state compression approach is the loss of RSVP's automatic adaptation to routing changes. Because refresh messages for each RSVP session follow the same path as data flow, RSVP reservation can automatically adapt to routing changes including multicast group membership changes. When a node compresses the RSVP sessions currently shared with a neighbor node to a single digest, however, RSVP loses the ability to trace down the paths of individual flows.

8.7 Summary

To improve the RSVP performance, we presented two changes to RSVP. We let each node compress aggregate RSVP state to a digest that can be carried in a single packet, which is then exchanged between neighbor RSVP nodes. The digest computation is done in a structured way, so that state inconsistency between two neighbors can be quickly located and repaired. This state compression approach considerably reduces the message overhead of RSVP. We also enhance RSVP with an acknowledgment option, so that lost messages can be quickly re-

transmitted. Our work suggests that the acknowledgment mechanism should be considered as a complement to, rather than a conflict with, soft-state protocol designs. Although reliable delivery of control messages cannot be used to replace soft-state refreshes, use of acknowledgment speeds up state synchronization in case of message losses.

CHAPTER 9

Comparison between Two State Compression Techniques

A good state compression technique is key to the previous two mechanisms. It should satisfy several requirements. Firstly, it should have a low bandwidth overhead and at the same time achieve a high level of state consistency. Secondly, it should have a short recovery time. Thirdly, it should have a low computation overhead and a low storage overhead.

There is generally a tradeoff between bandwidth overhead and state consistency. For example, a smaller Bloom-filter digest may produce more false positives, resulting in a lower level of state consistency. There is also a tradeoff between the bandwidth overhead of the *detection* phase and that of the *recovery* phase. Some techniques may have a higher detection overhead, but a lower recovery overhead. One needs to take into consideration all the different tradeoffs when selecting a state compression technique.

In this chapter, we evaluate and compare the two state compression techniques presented in Chapter 7 and Chapter 8. We identify the different tradeoffs provided by each technique. Protocol designers can choose one of the two techniques or they can design their own state compression technique to satisfy their specific requirements.

9.1 Methodology

To make the comparison meaningful, we apply both techniques to BGP and evaluate how well they can correct routing inconsistencies. Following the methodology in Chap 7, we simulate two BGP peers R_A and R_B and introduce random errors into R_B's $RibIn$. We then measure the error recovery ratio and bandwidth overhead of each technique. We generate four types of errors: insertion, removal, modification and mixed errors (a combination of the previous three types of errors). The BGP table used in this study has 101,404 routes.

To compute a Bloom Filter digest, we first compute a 128-bit MD5 hash over each route and then choose three 13-bit values as the hashes of the route. We use a digest size of 1024 bytes and an encoding ratio (α) of 5 or 8. Because the two encoding ratios produce quite different results in some cases, we present both of their results.

To compute a digest using the Digest Tree technique, we use a 2-level or a 3-level tree structure with a branching factor between 64 and 110. The hashes of the tree nodes are computed using CRC32. Since the results are similar for the two tree levels and the different branching factors, we present only those for the 2-level tree with a branching factor of 110.

9.2 Error Recovery Ratio and Bandwidth Overhead

Figure 9.1 compares the error recovery ratio of three instances of the two state compression techniques: (a) Digest Tree with a branching factor of 110, (b) Bloom Filter with an encoding ratio of 5, and (c) Bloom Filter with an encoding ratio of 8. To simplify our discussion, we denote these three instances as DTree, BFilter-5 and BFilter-8 respectively.

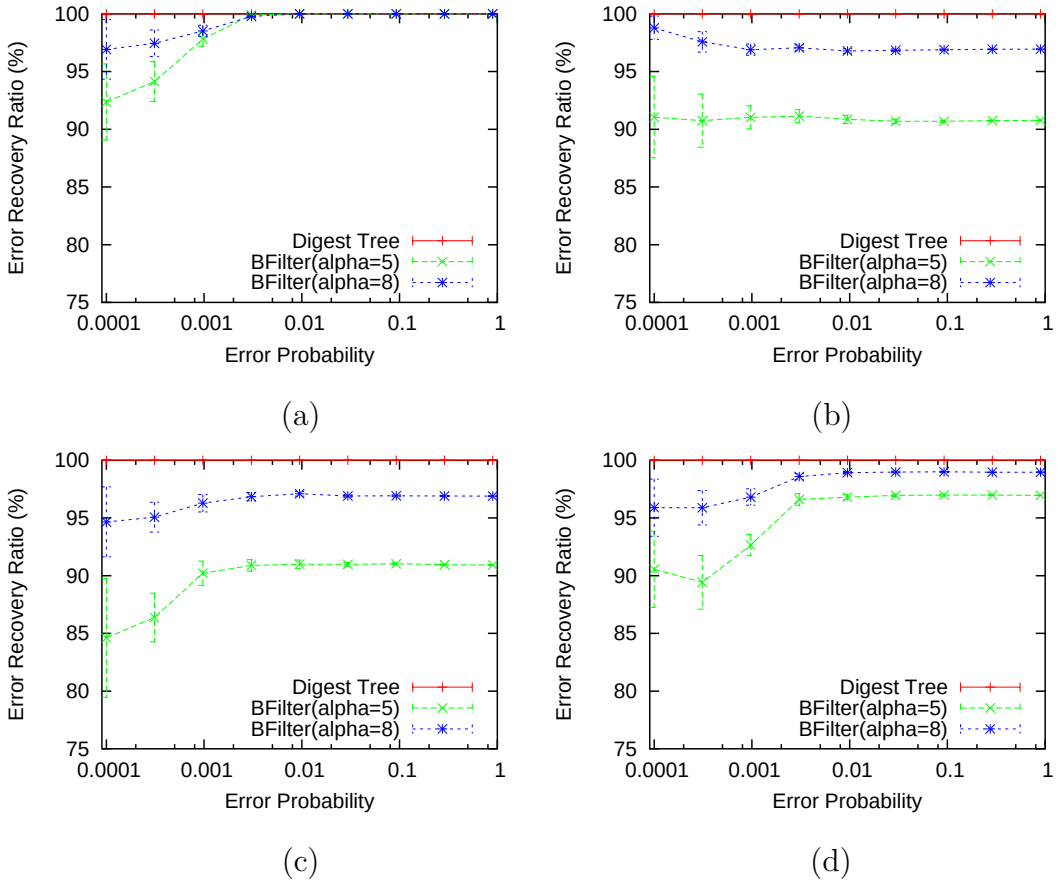


Figure 9.1: Error Recovery Ratio: (a) removal errors; (b) insertion errors; (c) modification errors; and (d) mixed errors.

First, it is clear that DTree has the highest error recovery ratio. In fact, DTree corrected all the errors in all of our experiments, i.e. its error recovery ratio is 100%, regardless of the error type and error probability. This is also true when we use a branching factor lower than 110 and a slightly higher tree structure. We will provide an explanation for this behavior later.

Secondly, we can see that the two curves corresponding to BFilter-5 and BFilter-8 overlap in the first figure when the error probability is high while they are disjoint in all the other figures. This is because different experiments test dif-

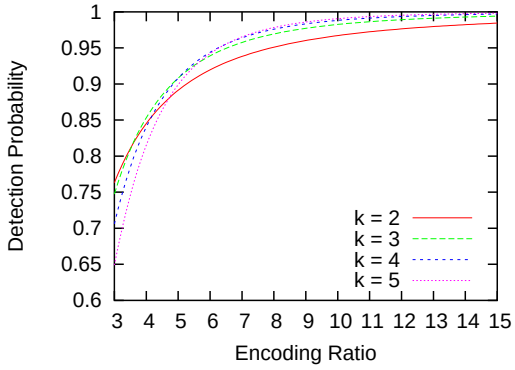


Figure 9.2: Probability of Detecting and Correcting a False Route using Bloom Filter-based Digest

ferent phases of this technique. In the experiment corresponding to Figure 9.1(a), we use only the "missing routes test". One characteristic of this test is that, as the probability of error P_e increases, the test becomes more accurate and the error recovery ratio also increases. When P_e rises above a certain threshold, the error recovery ratio becomes 100% for both BFilter-5 and BFilter-8. Section 7.7.1 provides more detailed explanation of this phenomenon.

Thirdly, Figure 9.1 shows that BFilter-8 has a higher error recovery ratio than BFilter-5 in most cases. As we have explained in Section 7.7.1, Bloom Filter with a higher encoding ratio has a lower false positive rate, which in general leads to a higher error recovery ratio. More specifically, given the encoding ratio and the number of hash functions used in the digest computation, the probability q of detecting and correcting a false route in most cases is approximately $1 - (1 - e^{-k/\alpha})^k$. We plot the relationship between k, α and q in Figure 9.2. As you can see, q increases with α for a given number of hash functions (k).

Below we provide more explanation of DTree's high error recovery ratio. Suppose we insert a false route into R_B's routing table. Let's designate the digest trees computed by the two peering routers $DTree_A$ and $DTree_B$. The Digest Tree

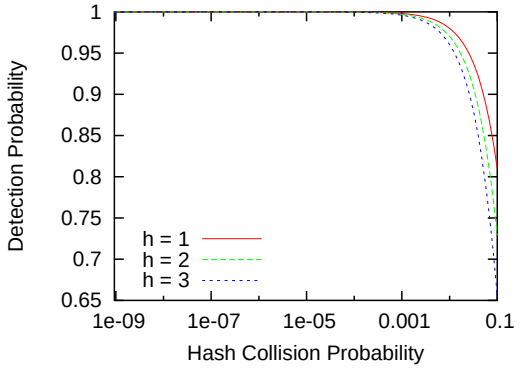


Figure 9.3: Probability of Detecting and Correcting a False Route using Digest Tree

technique can detect the false route only if $DTree_A$ and $DTree_B$ have different hashes in all the tree nodes from their root to the particular leaf node. Suppose the hash function has a collision probability of p and the tree has h levels. Assuming that the hashes are independent from each other, the probability that R_B will detect and correct the false route is $q = (1-p)^{h+1}$. Note that the route itself has a hash so there are $h+1$ hash comparisons. We plot q using different p and h in Figure 9.3. As shown in the figure, lower p and h can both result in a higher detection probability. In our experiment, we use a two-level digest tree and the collision probability of CRC32 is 2^{-32}, so the detection probability is very close to 1.

In Figure 9.4, we compare the bandwidth overhead of Digest Tree and Bloom Filter-based Digest. For reference, we also include the bandwidth overhead of a full BGP table Exchange (it shows the overhead of the traditional soft-state mechanism). First, we can see that BFilter-5 and BFilter-8 incur similar bandwidth overhead. Second, in Figure9.1(a), (c) and (d), the curve for DTree is lower than the other two curves for low error probabilities, but it eventually rises above the other two curves at an error probability between 0.01 and 0.1. This

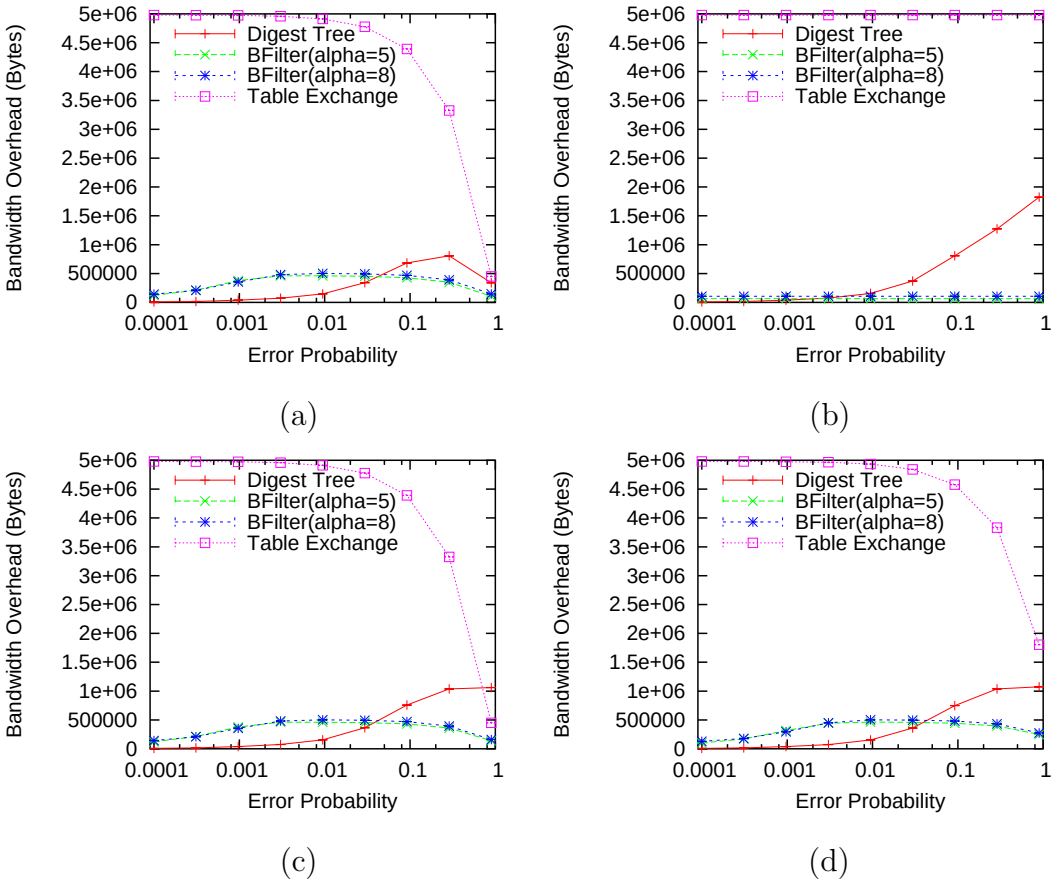


Figure 9.4: Bandwidth Overhead: (a) removal errors; (b) insertion errors; (c) modification errors; and (d) mixed errors.

is because multiple routes can be hashed to a slot in the hash table. When the error probability increases, it is more likely that at least one route in a hash slot is false. However, in order to identify this one false route, the two peer routers need to compare the hash of every route in that hash slot and each comparison incurs five to eight bytes of overhead (an address prefix + a 32-bit hash). As the error probability approaches 1, the bandwidth overhead will eventually include the hashes in every tree node and the hash of every route.

We also note that Figure 9.4(b) is different from the other figures. First, the

curves for BFilter-5, BFilter-8 and TableExchange are all flat. We have explained this phenomenon in Section 7.7.2. Secondly, the curve for DTree has a higher tail in this experiment than in the other experiments. This behavior reflects an implementation decision: we let R_B start the recovery process in order to reduce the recovery time. As a result, R_B needs to send the hashes of its routes to R_A in a later step. Since the error type is insertion, R_B will have more inserted routes and thus a higher bandwidth overhead when the error probability is higher. Fortunately, the experiments with removal or modification errors do not have this problem.

9.3 Recovery Time

We define recovery time as the period of time from sending the first digest message to receiving the message that corrects the last error. The specific recovery time depends on the round-trip time, the message processing time and several implementation details. If we assume that the round-trip time (T) is the dominant factor, then the recovery time of Digest Tree is $(h+1)/2 \cdot T$ and the recovery time of Bloom Filter-based Digest is at most $3/2 \cdot T$. Because $h \geq 2$, the Bloom Filter-based Digest technique usually recovers faster than or as fast as the Digest Tree technique.

9.4 Computation Overhead and Storage Overhead

Now we estimate the computation overhead of the two techniques. Suppose we want to insert a route to a digest tree. We need to first compute a hash for this route and then update all the hashes from the leaf node (i.e. the hash slot associated with the route) to the tree root. Therefore, we need to compute $h+1$

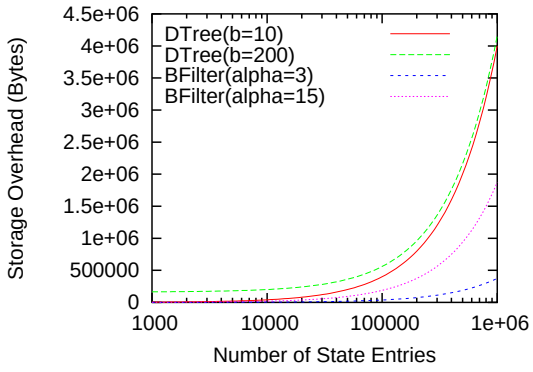


Figure 9.5: Comparison of Storage Overhead

hashes. On the other hand, Bloom Filter-based Digest requires computing k hashes. Since both h and k are small numbers, the two techniques should have comparable computation overhead if they use the same hash function.

Finally, we compare the storage overhead of these two techniques. Suppose there are n routes. If each hash is s bytes and each tree node has b children, the storage overhead of Digest Tree is $n \cdot s + b \cdot (b^h - 1)/(b - 1) \cdot s$. The storage overhead of Bloom Filter-based Digest is $n \cdot \alpha$. In Figure 9.5, the two top curves correspond to the storage overhead of Digest Tree with the branching factor 10 and 200 respectively. Both are obtained using a two-level tree structure and a hash size of 4 bytes. The two lower curves correspond to the storage overhead of Bloom Filter-based Digest with the encoding ratio of 5 and 15 respectively. These curves give the typical range of storage requirement that an implementation may have. We can see that the storage overhead of Digest Tree is generally higher than that of Bloom Filter-based Digest.

9.5 Summary

In summary, each of the two state compression techniques has its own pros and cons. Digest Tree typically corrects more errors than Bloom Filter-based Digest. However, Digest Tree has a higher bandwidth overhead when the error probability is high. In addition, Bloom Filter-based Digest requires less storage and recovers faster. Therefore, which technique is more preferable depends on many factors: the state consistency requirement, the minimum available bandwidth, the typical error type, the expected error probability and storage availability. The protocol designer needs to take into consideration all the requirements to choose or design the right technique. For example, if the error probability may vary within a wide range as is typical in a large and heterogeneous network, then Bloom Filter-based Digest may be a better choice than Digest Tree when low bandwidth overhead is the main requirement.

CHAPTER 10

Two Scalable Approaches to Building Resilient Network Protocols

A resilient protocol should be able to handle unexpected failures in a scalable fashion. This dissertation proposes two novel approaches *Lightweight Preventive Detection (LPD)* and *Persistent Detection and Recovery (PDR)* that help protocol designers to achieve this goal. In the previous chapters, we have presented three protocol mechanisms that follow these two approaches. More specifically, we presented in Chapter 6 an LPD mechanism that enables BGP to detect false BGP updates for certain important Internet sites efficiently. In Chapter 7 and Chapter 8, we presented two PDR mechanisms that allow BGP and RSVP to detect and recover from undetected state inconsistencies. In this chapter, we further discuss the rationale behind the proposed approaches as well as their limitations.

10.1 Lightweight Preventive Detection

Unlike traditional security mechanisms, our LPD approach does not rely on cryptography to detect messages containing false information, but instead makes use of certain properties of the network infrastructure. For example, our Adaptive Path-Filtering mechanism protects important Internet services against route-

hijacking by exploiting two common characteristics of these services. Such mechanisms usually incur low computation and bandwidth overhead, so they can be used for real-time detection in high-speed networks. Moreover, they typically require minimal changes to the current infrastructure and are thus readily deployable in the Internet.

To show the generality of this approach, we describe another LPD mechanisms developed by us. We also describe a mechanism developed by Subramanian et al that follows a similar approach. We then discuss the limitations of our approach and its relationship with strong validation mechanisms.

10.1.1 Example 1: Detection of Invalid MOAS Conflicts

In [ZPW02], we proposed a mechanism that exploits the rich connectivity in the Internet to detect invalid MOAS (Multiple Origin AS) conflicts. When one address prefix is originated by multiple ASes, it is said to be involved in a MOAS conflict. Some MOAS conflicts reflect legitimate operational practices such as multi-homing, but others are caused by misconfigurations and route hijacking. However, BGP provides no mechanism to distinguish invalid MOAS conflicts from valid ones. To solve this problem, we let each router attach a set of "legal origin ASes" (called a MOAS list) to its BGP announcement when it originates a prefix. A router can detect an invalid MOAS conflict when it receives BGP announcements with different MOAS lists for the same address prefix. This mechanism is viable in a richly connected network, since isolated attackers or misconfigured routers may block some but not all the valid BGP announcements. Note that this simple mechanism does not try to address all the possible BGP faults and attacks – it detects only false BGP routes with invalid origin ASes.

Figure 10.1 shows an example of how our scheme can be used to detect an

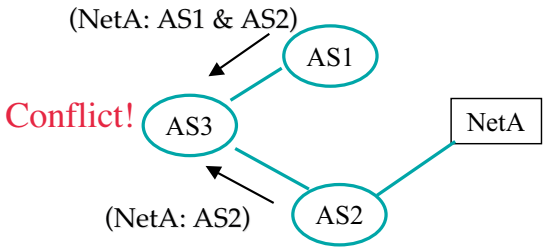


Figure 10.1: An Example of Using MOAS List to Detect Invalid MOAS Conflict (AS1 is an invalid origin AS of NetA)

invalid MOAS conflict. In this example, $AS1$ is an attacker that atempts to hijack NetA, while $AS2$ is an upstream ISP of NetA. When $AS2$ originates the address prefix of NetA, its announcement includes the MOAS list ($AS2$). The attacker also originates NetA, but the MOAS list in its announcement includes both $AS1$ and $AS2$. When both BGP announcements reach $AS3$, $AS3$ can detect a conflict in their MOAS lists and it can further investigate which announcement is bogus. Note that it is possible for $AS2$ to misconfigure its MOAS list. However, as long as the misconfigured list is not identical to $AS1$'s list, $AS3$ can still detect a conflict.

We simulated this mechanism in SSFNET [SSF] using sampled Internet topologies. More specifically, we derived an AS topology from an archived routing table and randomly selected a set of ASes from this topology. The sampled topology then includes all the selected ASes and their adjacencies.

In a 46-AS topology, our mechanism can reduce the percentage of ASes adopting the falsely originated route from 36% to 0.15%, when 4% of the AS's are compromised. When the percentage of compromised ASes increases to 30%, still only 9.8% of the remaining ASes adopt the false route, compared to 51% without our mechanism. The results are even better in a 63-AS topology. Our simulation further shows that even partial deployment of this mechanism can mitigate the

effect of falsely originated routes. Due to the constraints of the SSFNET simulator, we could not simulate large topologies. In general, the effectiveness of our mechanism depends on the *richness* of the topology, which seems to be positively correlated with the size of the topology. But the verification of this correlation remains a future research area.

10.1.2 Example 2: Whisper

Recently, Subramanian et al. proposed the Whisper protocol for the detection of invalid AS paths [SRS04]. Although they address a different problem, their basic approach is very similar to ours – they also exploit rich network connectivity to detect BGP routes with conflicting information. More specifically, each BGP route includes a one-way hash that is incrementally computed by every router along the path using its AS number. When a router receives two BGP routes to the same address prefix, it checks whether their hash values are consistent, i.e. whether they satisfy a mathematical condition. If one of the routes is false and the other is valid, their hash values will be inconsistent and the router raises an alarm. Therefore, as long as an adversary does not block every route to a destination, its false route will raise an alarm when the route reaches a router that already has a valid route to the destination.

10.1.3 Discussion

Strong validation mechanisms, such as SBGP, are accurate in detecting false information, but they usually incur a high computation overhead and therefore do not scale well. In addition, some of them require a public key infrastructure that is not available today. On the other hand, LPD mechanisms are fast and readily deployable, but they also have limitations. For example, the mechanism

described in Section 10.1.1 can detect an invalid MOAS conflict when two routes have different MOAS lists, but it cannot identify which route carries the false information. The Whisper protocol has the same limitation. Therefore, a strong validation mechanism is still required in this scenario. This validation mechanism could be cryptography-based if one is available or it could simply be a manual check. In summary, LPD mechanisms can serve as a front-end to more accurate validation mechanisms. The benefit of this two-tier design is that it eliminates the need to do expensive validation on *every* message, thus improving the overall protocol efficiency.

Note that a mechanism that exploits network properties to detect false information can be considered an LPD mechanism only if its overall performance is scalable. For example, although the Whisper protocol does not use public-key cryptography, it makes use of *one-way hashes* such as SHA [Sch96]. Therefore its scalability depends on the overhead of generating, updating and verifying the hashes. Whether this protocol can be deployed successfully depends on whether the hashing operations are fast enough to accommodate the routing dynamics in the current and future Internet [1].

10.2 Persistent Detection and Recovery

Our Persistent Detection and Recovery approach allows network nodes to recover from *state inconsistencies* without incurring per-state refresh overhead. The key idea behind this approach is *state compression* – every node compresses its entire state space into a digest and the digest can be used to quickly identify any state

[1] Whisper seems to be able to handle current levels of routing dynamics. According to [SRS04], Whisper can update and verify over 100, 000 routes per minute if 1024-bit keys are used. Generating a hash may take more than 1 second, but this operation is performed only once over many days.

inconsistencies. Our approach also uses periodic digest exchanges to provide continuous protection against any potential faults or attacks.

In this section, we first elaborate on the rationale behind our approach. We then discuss the limitations of our approach and clarify the relationship between our work and two other areas of work.

10.2.1 Rationale behind Our Approach

10.2.1.1 Reason for Periodic Refreshes

Our approach follows the soft-state paradigm of network state management – it uses periodic digest messages to achieve long-term state consistency. The rationale behind the persistent refreshes is, as we have explained in Section 2.2 and 7.2.2, that they enable a protocol to recover from unexpected failures such as memory corruption and human errors. However, there has been much debate over whether soft state or hard state is a better approach in the past. Most notably, [JGK03] compared several variations of the basic soft-state mechanism with a hard-state mechanism. They find that explicitly acknowledging trigger messages allows the soft-state approach to achieve comparable (and sometimes better) consistency than that of the hard-state approach. We agree with the first part of their finding: acknowledging trigger messages can improve the state consistency of a soft-state mechanism (this is why we proposed adding acknowledgment to RSVP in [WTZ99]). However, their conclusion that the hard-state approach can achieve a higher state consistency than all the other soft-state variations is based on a limited fault model. This fault model does not include faults and attacks that can remove, insert or modify a *receiver*'s state. We believe that a resilient protocol should be able to handle *any* fault or attack, therefore we choose the soft state approach.

10.2.1.2 Reasons for State Compression and Receiver Participation

Our approach is different from the traditional soft-state mechanism. The latter suffers from high refresh overhead due to two design characteristics. First, the traditional soft-state mechanism treats each piece of state separately and therefore it incurs per-state overhead. Second, it is open-loop, i.e. the sender and the receiver do not cooperate to identify the inconsistent state. In contrast, our approach achieves efficiency by (a) compressing the state entries into a more compact form; and (b) allowing the receiver to participate in the detection and recovery process.

10.2.1.3 Reasons for Not Tuning Refresh Timer

All the existing refresh overhead reduction mechanisms still preserve the two characteristics of the traditional soft-state mechanism ([HSC95, SEF97, PS97, RS98]). They then try to reduce the overhead by adjusting the refresh interval in various ways (see Section 2.2.3 for more details on these mechanisms). For example, in order to maintain a constant refresh overhead, the Scalable Timer mechanism [SEF97] increases the refresh interval when there is more state in a protocol. Although the protocol is still able to recover from unexpected state inconsistencies, the longer refresh interval means that it will take longer for neighboring nodes to reach consistency in case of unexpected attacks or faults. If we measure *state consistency* using the percentage of time when two peers have consistent state information (following [RM99] and [JGK03]), increasing the refresh interval will result in lower state consistency.

Specifically, if we assume that faults and attacks strike a state entry e at a rate of λ and refresh messages are sent at a rate of μ (both arrivals follow a Poisson process), we can use a simple Markov chain to model this scenario. The Markov

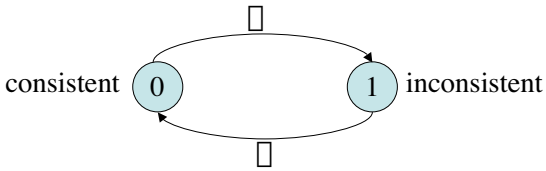


Figure 10.2: Markov Chain for Modeling State Consistency between Two Nodes (λ = arrival rate of faults and attacks, μ = arrival rate of refreshes)

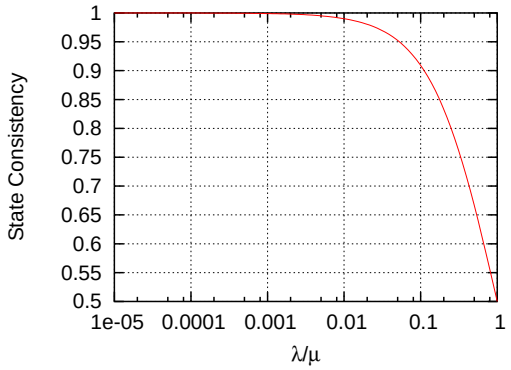


Figure 10.3: State Consistency as a Function of λ/μ (λ = arrival rate of faults and attacks, μ = arrival rate of refreshes)

chain has only two modes 0 (consistent) and 1 (inconsistent). It transits from 0 to 1 with a rate of λ and from 1 to 0 with a rate of μ. The percentage of time when two peers have a consistent value for e is $p_0 = \mu/(\lambda + \mu) = 1/(\lambda/\mu + 1)$. This result means that, if we increase the refresh interval, λ/μ will increase and the state consistency will decrease.

To further illustrate the effect of increased refresh interval, we plot p_0 in Figure 10.3. When λ/μ is 0.01, p_0 is around 0.99 (the state is consistent 99% of the time). Now suppose the number of state entries is increased by a factor of 10. In order to keep the refresh overhead constant, we need to increase the refresh interval by 10 times. λ/μ then becomes 0.1 and p_0 will drop to 0.909. If the number of state entries is increased by another factor of 10, the state consistency

will further drop to 0.5.

However, a common assumption made by these mechanisms is that unexpected faults or attacks rarely occur, i.e. $\lambda/\mu \approx 0$. Therefore, increasing the refresh interval will not significantly reduce the state consistency. We do not make this assumption in our design for robustness considerations.

Practically speaking, even if the state consistency will not decrease much by increasing the refresh interval, there is often an upper bound on the interval that is imposed by the specific application. For example, a VoIP call cannot tolerate a delay of ten minutes for the signaling protocol to repair a corrupted reservation. This is a problem neither the Scalable Timer nor the Staged Timer mechanism can handle well. For example, the Staged Timer mechanism suggests that RSVP increase its refresh timer exponentially for an RSVP reservation after the reservation is set up and after any state change has been acknowledged. As a result, the refresh interval could potentially become very large, e.g. on the order of minutes or even hours, and the recovery latency will be quite long when anything unexpected occurs.

For the above reasons, we do not directly tune the refresh interval in our approach. However, timer adjustment mechanisms can still complement our approach as long as the combination of these different mechanisms can still satisfy the requirements for state consistency and recovery latency.

10.2.2 Discussion

The proposed PDR mechanisms are designed for BGP and RSVP. In both protocols, there may be a large amount of state shared between neighboring nodes and each node usually has a small number of neighbors. We have not applied PDR to the scenario where each node broadcasts its state to all the other nodes. In this

case, we may need to address the issue of a feedback explosion. In addition, we have not studied the scenario where the state on each node changes frequently. The challenge here is to ensure that the recovery process does not interfere with the propagation of new state changes.

Now we would like to clarify the relationship between our work and another area of work. Raman and McCanne studied announce-listen protocols in [RM99]. They call the announce-listen type of communication ([CRS98, FJM95]) "soft state-based communication". Announce-listen protocols are usually used for data delivery, for example, distribution of multicast session information [Han96]. These protocols assume that, once a piece of data is received, it will remain correct. Therefore, the subsequent announcements serve only one purpose, that is, to reach new receivers. This explains why they consider the transmissions of already consistent data items mostly "redundant" and try to assign lower priority to these redundant messages. This assumption is obviously not valid for our case, and thus the two-level priority scheme they proposed would not improve state consistency.

CHAPTER 11

Conclusion and Future Work

Network faults and malicious attacks are increasingly threatening the availability of Internet services. It is therefore important for Internet protocols to function well under adverse conditions. As the Internet continues to grow, however, this goal becomes more difficult to achieve since protocols often need to maintain more state and there are potentially more sources of faults and attacks. In this dissertation, we propose two scalable approaches to improving Internet protocols' ability to withstand unexpected failures.

The first approach, Lightweight Preventive Detection (LPD), exploits network properties such as routing stability to quickly detect suspicious information. The second approach, Persistent Detection & Recovery (PDR), assumes that undetected failures will occur. To recover from the state inconsistencies resulting from such failures, PDR periodically checks the consistency between neighboring network nodes and uses state compression techniques to efficiently correct any inconsistencies.

We have applied our approaches to one critical component of the Internet infrastructure – BGP routing that provides the data delivery paths in the global Internet. Our Adaptive Path-Filtering mechanism protects the BGP routes to top-level DNS servers. Our Fast Routing Table Recovery (FRTR) mechanism can correct a variety of routing inconsistencies that cannot be handled by the current BGP design. Furthermore, it can reduce BGP's failure recovery overhead

by one to two orders of magnitude. We have also applied the PDR approach to make RSVP recover from state inconsistencies more efficiently.

Any truly resilient system must include multiple protection fences since no single fence can be strong enough to defend against all potential faults and attacks. The proposed solutions will not protect all protocols under all types of adverse conditions, but will add to the overall resilience of the system.

In my future work, I want to study how LPD and PDR can be applied to link-state routing protocols such as OSPF. I also plan to investigate the possibility of using link-state for inter-domain routing. Compared with the current path-vector protocol BGP, the advantage of a link-state protocol is that it can provide multi-path routing, faster convergence, easier debugging and better security. However, the pure link-state approach requires propagating routing policies globally, while ISPs prefer to keep their policies known to their peers only. Therefore, a hybrid approach that combines link-state and path-vector may be necessary.

Futhermore, I am interested in improving measurement infrastructures and inference techniques, since a solid understanding of how the Internet actually performs is the foundation for improving Internet resilience. It is not difficult to collect a large amount of data on Internet performance, but measurement studies without a good methodology often provide misleading results [Pax01]. For example, my analysis of BGP routing data revealed several problems in the public routing measurement infrastructures that could lead to false conclusions [WZP02]. I plan to continue my work in this area.

Finally, following my study on BGP's behavior during the Nimda worm attack, I would like to use measurement data and large-scale network simulation to study how other critical Internet services such as DNS behave under DDoS attacks. Moreover, since DDoS attacks can cause significant damage in a short

period of time (e.g. disable a large number of servers), I plan to study how to ensure that Internet services can recover quickly from these attacks.

References

[ALM02] R. Arends, M. Larson, D. Massey, and S. Rose. "DNS Security Introduction and Requirements." *Work in Progress*, December 2002.

[BBL98] T. Bates, R. Bush, T. Li, and Y. Rekhter. "DNS-based NLRI Origin AS Verification in BGP." *Work in Progress*, 1998.

[BCM02] J. Byers, J. Considine, and M. Mitzenmacher. "Fast approximate reconciliation of Set Differences." Technical Report 2002-019, Boston University Computer Science Department, 2002.

[BGS00] L. Berger, D. Gan, G. Swallow, P. Pan, and F. Tommasi. "RSVP Refresh Overhead Reduction Extensions." *Work in Progress*, June 2000.

[Bil98] Per Gregers Bilse. "Re: Is Qwest leaking routes?" `http://www.merit.edu/mail.archives/nanog/1998-10/msg00841.html`, October 1998.

[Blo70] B. H. Bloom. "Space/time Trade-offs in Hash Coding with Allowable Errors." *Communications of the ACM*, **13**(7):422–426, 1970.

[BM02] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters." In *Proceedings of the 40th Annual Allerton Conference on Communication, Control and Computing*, October 2002.

[Bon97] Vincent J. Bono. "7007 Explanation and Apology." `http://www.merit.edu/mail.archives/nanog/1997-04/msg00444.html`, April 1997.

[BZ97] R. Braden and L. Zhang. "Resource ReSerVation Protocol (RSVP), Version 1 Message Processing Rules." *RFC 2209*, September 1997.

[BZB97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. "Resource ReServation Protocol (RSVP) – Version 1 Functional Specification." *RFC 2205*, September 1997.

[CER] CERT/CC. "CERT/CC Statistics." `http://www.cert.org/stats/`.

[CER02] CERT/CC. "Overview of Attack Trend." `http://www.cert.org/archive/pdf/attack_trends.pdf`, April 2002.

[CGH02] D.-F. Chang, R. Govindan, and J. Heidemann. "An Empirical Study of Router Response to Large BGP Routing Table Load." In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2002*, November 2002.

[Cha97] Ravi Chandra. "Re: Andrew Partan: Re: MAI.net filters." `http://www.merit.edu/mail.archives/nanog/1997-04/msg00480.html`, April 1997.

[Cis01] Cisco Systems. "Dealing with mallocfail and High CPU Utilization Resulting from the "Code Red" Worm." `http://www.cisco.com/warp/public/63/ts_codred_worm.shtml`, October 2001.

[Cla88] D. D. Clark. "The Design Philosophy of the DARPA Internet Protocols." In *Proceedings of the ACM SIGCOMM '88*, pp. 106–14, Stanford, CA, August 1988.

[Com] Computer Economics. `http://www.computereconomics.com/`.

[COP01] J. Cowie, A. Ogielski, B. J. Premore, and Y. Yuan. "Global Routing Instabilities Triggered by Code Red II and Nimda Worm Attacks." Technical report, Renesys Corporation, December 2001.

[CR03] E. Chen and Y. Rehkter. "Cooperative Route Filtering Capability for BGP-4." *Work in Progress*, February 2003.

[CRS98] M. Chandy, A. Rifkin, and E. Schooler. "Using Announce-Listen with Global Events to Develop Distributed Control Systems." *Concurrency: Practice and Experience*, **10**(11-13):1021–7, February 1998.

[Dan99] Sanjay Dani. "Heads up: 1280_7374_ Prefixed Route Leaks." `http://www.merit.edu/mail.archives/nanog/1999-04/msg00032.html`, April 1999.

[DEF94] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. "An Architecture for Wide-Area Multicast Routing." *Computer Communication Review*, **24**(4):126–35, October 1994.

[Del02] Michelle Delio. "How and Why the Internet Broke." `http://www.wired.com/news/technology/0,1282,55580,00.html`, October 2002.

[Dij74] E. W. Dijkstra. "Self-stabilizing Systems in spite of Distributed Control." *Communications of the ACM*, **17**(11):643–4, November 1974.

[Dil98] Michael Dillion. "Hunting for bogus BGP announcement for 204.106.93.155." `http://www.merit.edu/mail.archives/nanog/1998-10/.html`, October 1998.

[Don99] Sean Donelan. "Re: Routing Loop between Qwest and Sprint?" `http://www.merit.edu/mail.archives/nanog/1999-05/msg00157.html`, May 1999.

[Don03] Sean Donelan. "Router Crash Unplugs 1m Swedish Internet Users." `http://www.merit.edu/mail.archives/nanog/2003-06/msg00491.html`, June 2003.

[Far01] James A. Farrar. "Re: C&W routing instability." `http://www.merit.edu/mail.archives/nanog/2001-04/msg00209.html`, April 2001.

[FCA00] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. "Summary Cache: a Scalable Wide-area Web Cache Sharing Protocol." *IEEE/ACM Transactions on Networking*, **8**(3):281–293, 2000.

[FJM95] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang. "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing." In *Proceedings of the ACM SIGCOMM '95*, Boston, MA, September 1995.

[GAG03] G. Goodell, W. Aiello, T. G. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin. "Working Around BGP: An Incremental Approach to Improving Security and Accuracy of Interdomain Routing." In *Proceedings of NDSS 2003*, February 2003.

[Gil02] V. Gill. "Lack of priority queuing on route processors considered harmful." In *NANOG 26*, October 2002.

[GM95] J. J. Garcia-Lunes-Aceves and S. Murthy. "A Loop-Free Path-Finding Algorithm: Specification, Verification and Complexity." In *Proceedings of the IEEE INFOCOM '95*, April 1995.

[Han96] M. Handley. "SAP: Session Announcement Protocol." *Work in Progress*, November 1996.

[Hef98] A. Heffernan. "Protection of BGP Sessions via the TCP MD5 Signature Option." *RFC 2385*, August 1998.

[HSC95] H. W. Holbrook, S. K. Singhal, and D. R. Chriton. "Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation." In *Proceedings of the ACM SIGCOMM '95*, pp. 328–341, 1995.

[HW01] K. Houle and G. Weaver. "Trends in Denial of Service Attack Technology." `http://www.cert.org/archive/pdf/DoS_trends.pdf`, October 2001.

[ICA02] ICANN. "DNS Security Update 1." `http://www.icann.org/committees/security/dns-security-update-1.htm`, January 2002.

[ICM02] G. Iannaccone, C. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. "Analysis of Link Failures in an IP Backbone." In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2002*, November 2002.

[Inc04] Sandvine Inc. "Worms Gobbling Broadband Profits: The Financial Impact of Attack Traffic on Service Provider Networks." `http://www.sandvine.com/`, February 2004.

[JGK03] P. Ji, Z. Ge, J. Kurose, and D. Towsley. "A Comparison of Hard-state and Soft-state Signaling Protocols." In *Proceedings of ACM SIGCOMM '03*, Karlsruhe, Germany, August 2003.

[KA98] S. Kent and R. Atkinson. "Security Architecture for the Internet Protocol." *RFC 2401*, November 1998.

[Ker04] Zeus Kerravala. "As the Value of Enterprise Networks Escalates, So Does the Need for Configuration Management." The Yankee Group Report, January 2004.

[KLM00] S. Kent, C. Lynn, J. Mikkelson, and K. Seo. "Secure Border Gateway Protocol (S-BGP) – Real World Performance and Deployment Issues." In *Proceedings of Network and Distributed System Security Symposium*, February 2000.

[KLS00] S. Kent, C. Lynn, and K. Seo. "Secure Border Gateway Protocol." *IEEE Journal of Selected Areas in Communications*, **18**(4), April 2000.

[Kro98] Bernhard Kroenung. "AS8584 Taking Over the Internet." `http://www.merit.edu/mail.archives/nanog/1998-04/msg00047.html`, April 1998.

[LAB00] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. "Delayed Internet Routing Convergence." In *Proceedings of the ACM SIGCOMM 2000*, August 2000.

[LAJ99] C. Labovitz, A. Ahuja, and F. Jahanian. "Experimental Study of Internet Stability and Wide-Area Backbone Failures." In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pp. 278–85, Madison, WI, June 1999.

[LMJ99] C. Labovitz, G. R. Malan, and F. Jahanian. "Origins of Internet Routing Instability." In *Proceedings of the IEEE INFOCOM '99*, pp. 218–26, New York, NY, March 1999.

[Mal98] G. Malkin. "RIP Version 2." *RFC 2453*, November 1998.

[MCI02] "WorldCom IP Network Operating Normally." MCI Press Release, October 2002.

[Mer] Merit. "Internet Routing Registry." `http://www.irr.net/`.

[MF02] O. Maennel and A. Feldmann. "Realistic BGP Traffic for Test Labs." In *Proceedings of ACM SIGCOMM 2002*, August 2002.

[MFR78] J. M. McQuillan, G. Falk, and I. Richer. "A Review of the Development and Performance of the ARPANET Routing Algorithm." *IEEE Transactions on Communications*, **26**(12):1802–1811, 1978.

[MJ98] G. R. Malan and F. Jahanian. "An Extensible Probe Architecture for Network Protocol Performance Measurement." In *Proceedings of the ACM SIGCOMM '98*, Vancouver, BC, Canada, September 1998.

[Moc87] P. Mockapetris. "Domain Names–Concept and Facilities." *RFC 1034*, November 1987.

[Moy98] J. Moy. "OSPF Version 2." *RFC 2328*, April 1998.

[MP03] D. McPherson and K. Patel. "Experience with the BGP-4 Protocol." *Work in Progress*, September 2003.

[MR01] D. Massey and S. Rose. "DNS Security Introduction and Requirements." *Work in Progress*, September 2001.

[MSB02] D. Moore, C. Shannon, and J. Brown. "Code-Red: a Case Study on the Spread and Victims of an Internet Worm." In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2002*, November 2002.

[Mur03] S. Murphy. "BGP Security Vulnerabilities Analysis." *Work in Progress, draft-ietf-idr-bgp-vuln-00.txt*, June 2003.

[MWA02] Ratul Mahajan, David Wetherall, and Tom Anderson. "Understanding BGP Misconfiguration." In *Proceedings of the ACM SIGCOMM*, August 2002.

[NIS04] NISCC. "Vulnerability Issues in TCP." `VulnerabilityIssuesinTCP`, May 2004.

[OGP03] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. "Why do Internet services fail, and what can be done about it?" In *4th Usenix Symposium on Internet Technologies and Systems (USITS'03),*, 2003.

[Pax01] V. Paxon. "Some Not-So-Pretty Admissions About Dealing With Internet Measurements (invited talk)." In *Workshop on Network-Related Data Management (NRDM 2001)*, 2001.

[Per88] R. Perlman. *Network Layer Protocols with Byzantine Robustness.* PhD thesis, MIT, October 1988.

[PS97] P. Pan and H. Schulzrinne. "Staged Refresh Timers for RSVP." In *Proceedings of the IEEE GLOBECOM '97*, pp. 3–8, Phoenix, AZ, November 1997.

[PZW02] D. Pei, X. Zhao, L. Wang, D. Massey, A. Mankin, S. Wu, and L. Zhang. "Improving BGP Convergence Through Consistency Assertions." In *Proceedings of the IEEE INFOCOM 2002*, June 2002.

[RIP] RIPE Routing Information Service Project. `http://www.ripe.net/ris/`.

[RIS] RISreport. `http://www.ris.ripe.net/risreport/`.

[Riv92] R. Rivest. "The MD5 Message-Digest Algorithm." *RFC 1321*, April 1992.

[RL95] Y. Rekhter and T. Li. "A Border Gateway Protocol (BGP-4)." *RFC 1771*, March 1995.

[RLH03] Y. Rekhter, T. Li, and S. Hares. "A Border Gateway Protocol 4 (BGP-4)." *Work in Progress, draft-ietf-idr-bgp4-23.txt*, November 2003.

[RM99] S. Raman and S. McCanne. "A Model, Analysis, and Protocol Framework for Soft State-based Communication." In *Proceedings of the ACM SIGCOMM '99*, pp. 15–25, Cambridge, MA, September 1999.

[rou] The Route Views Project. `http://www.antc.uoregon.edu/route-views/`.

[RS98] J. Rosenberg and H. Schulzrinne. "Timer Reconsideration for Enhanced RTP Scalability." In *Proceedings of the IEEE INFOCOM '98*, March 1998.

[RWX02] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. "BGP Routing Stability of Popular Destinations." In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2002*, November 2002.

[SAN01] SANS Institute. "Nimda Worm/Virus Report." `http://www.incidents.org/react/nimda.pdf`, October 2001.

[Sch96] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., 1996.

[SEF97] P. Sharma, D. Estrin, S. Floyd, and V. Jacobson. "Scalable Timers for Soft State Protocols." In *Proceedings of the IEEE INFOCOM '97*, pp. 222–9, Kobe, Japan, April 1997.

[SG96] B. Smith and J. J. Garcia-Luna-Aceves. "Securing the Border Gateway Routing Protocol." In *Proceedings of the IEEE Global Internet '96*, November 1996.

[SMW02] N. Spring, R. Mahajan, and D. Wetherall. "Measuring ISP Topologies with Rocketfuel." In *Proceedings of the ACM SIGCOMM '02*, Pittsburg, PA, August 2002.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. "End-To-End Arguments in System Design." *ACM Transactions on Computer Systems*, **2**(4):277–288, November 1984.

[SRS04] L. Subramanian, V. Roth, I. Stoica, S. Shenker, and R. H. Katz. "Listen and Whisper: Security Mechanisms for BGP." In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI'04)*, March 2004.

[SSF] The SSFNET Simulator. `http://www.ssfnet.org/`.

[SVK00] A. Shaikh, A. Varma, L. Kalampoukas, and R. Dube. "Routing Stability in Congested Networks: Experimentation and Analysis." In *Proceedings of the ACM SIGCOMM 2000*, pp. 163–74, Stockholm, Sweden, September 2000.

[Tel] Telstra. "BGP Data." `http://www.telstra.net/ops/bgptable.html`.

[VCG98] C. Villamizar, R. Chandra, and R. Govindan. "BGP Route Flap Damping." *RFC 2439*, November 1998.

[Whi04] R. White. "Architecture and Deployment Considerations for Secure Origin BGP (soBGP)." *Work in Progress, draft-white-sobgparchitecture-00.txt*, May 2004.

[WMP04] L. Wang, D. Massey, K. Patel, and L. Zhang. "FRTR: A Scalable Mechanism for Global Routing Table Consistency." In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2004.

[WTZ99] L. Wang, A. Terzis, and L. Zhang. "A New Proposal for RSVP Refreshes." In *Proceedings of IEEE ICNP '99*, pp. 163–72, Toronto, Canada, November 1999.

[WZP02] L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. Wu, and L. Zhang. "Observation and Analysis of BGP Behavior under Stress." In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2002*, November 2002.

[WZP03] L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, and L. Zhang. "Protecting BGP Routes to Top-Level DNS Servers." *IEEE Transactions on Parallel and Distributed Systems*, September 2003.

[Yu] J. Yu. "A Route-Filtering Model for Improving Global Internet Routing Robustness." `http://www.iops.org/Documents/routing.html`.

[Zeb] Zebra Routing Software. `http://www.zebra.org`.

[ZMW03] X. Zhao, D. Massey, S. F. Wu, M. Lad, D. Pei, L. Wang, and L. Zhang. "Understanding BGP Behavior through a Study of DoD Prefixes." *Proceedings of DISCEX 2003*, April 2003.

[ZPW01] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. Wu, and L. Zhang. "An Analysis of BGP Multiple Origin AS (MOAS) Conflicts." In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2001*, November 2001.

[ZPW02] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. Wu, and L. Zhang. "Detection of Invalid Routing Announcements in the Internet." In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002.