CHAPTER 1

Introduction

1.1 Motivation and Goal

Today's Internet is suffering from various DDoS attacks [1], such as recent attacks to large Internet sites including Yahoo and eBay. These DDoS attacks have posed a threat to Internet security. Sites under attack cannot provide services to customers during the attack period, which causes a large loss of money.

One possible method for stopping such attacks is to trace back to identify the attackers and shut off those links. To achieve this goal, one option is to check the source address field in the IP header of packets. Unfortunately, the existing Internet protocols allow senders to forge source addresses arbitrarily, and receivers have no way to tell if packets come from the real source addresses. Commonly, DDoS attackers use such kinds of IP spoofing to hide their true identities and hinder the tracing of their malicious packets.

IP spoofing itself can be a kind of malicious attack. Remember, receivers will use the source address of the packet as the destination of reply packets, so that attackers can assault two sites at the same time, since any bad responses will be sent to the forged sender indicated in the source address field.

We believe the solution to the problems is to validate the source address of each

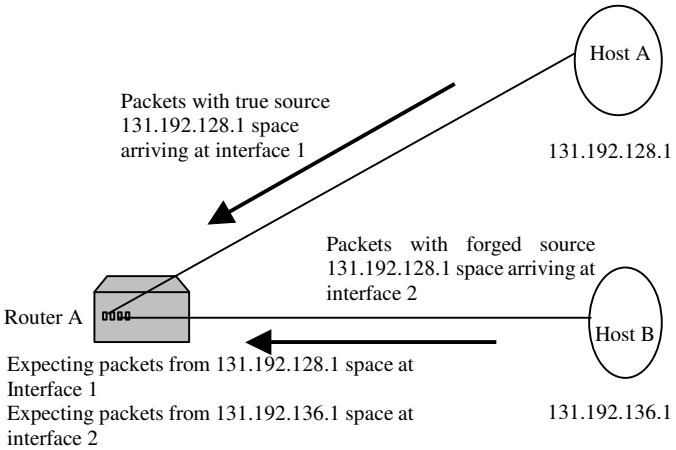


Figure 1. An example of source validation scheme.

arriving packet at routers. Based on the knowledge of the correct incoming interface for packets from a specific source address, a router can filter out those packets that don't arrive at the expected incoming interface. Figure 1 is an example of this scheme.

In Figure 1, router A has the knowledge that packets with source 131.192.128.1 should come in from the interface 1, so it will accept the packet from host A and drop all bad packets with forged source address from host B. If all routers on the Internet have this functionality, attackers cannot spoof source addresses, and all traceback mechanisms can operate with confidence

Source validation has many other advantages. On the Internet, many services rely on accurate source addresses. For example, some fair queuing algorithms consider source addresses as a factor to determine bandwidth allocation. In some QoS protocols such as RSVP [2], source addresses are used to make policy control decisions for resource reservation. Here accurate source information will help to prevent abuse of resource.

To validate source addresses, the key problem is how to know which incoming interface is correct for packets from a specific source. Currently, many methods are proposed to check source addresses of arriving packets. The following Section briefly discusses several related works and their shortcomings.

1.2 Related Works

1.2.1 Filtering

Filtering is a preventive approach to IP spoofing attacks. The general approach of filtering has been proposed in [3]. The following paragraphs discuss two main filtering mechanisms: *symmetric filtering* [3] and *periphery filtering* [4].

As we know, the forwarding table at each router indicates the interface that should be used to deliver a packet to a particular destination address D. The most obvious method to determine the correct incoming interface for packets from source address D is *symmetric filtering*, which reverses the forwarding table and considers the outgoing interface to D as the incoming interface from D. This method assumes that symmetric routing is used to deliver packets, which means packets from the source should follow the same interface as the router uses to sends packets to it. In such cases, symmetric filtering works properly and efficiently. However, asymmetric routing is common in the

Internet. With asymmetric routing, packets from a source follow a route different from the route used to send packets to that source address. Asymmetric routing is especially common in inter-domain routing. Figure 2 provides a simple example.

As Figure 2 shows, router C will follow interface 1 to destinations 131.192.136.0/24, but packets from that address space should arrive at interface 2. In this case, symmetric filtering is infeasible, because it will drop many packets even though they have valid source addresses.

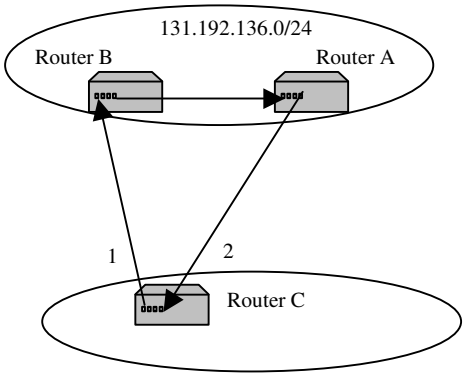


Figure 2. An example of asymmetric routing.

Another method to validate source addresses is *periphery filtering*, in which routers at the border of a network ensure that incoming packets have source addresses from outside, and outgoing packets have source addresses from inside. For instance, if we deploy such filtering in Figure 2, router A should make sure all incoming packets don't have source addresses 131.192.136.0/24; on the other hand, router B will drop all packets with outside source addresses.

This method can prevent a source address space from being forged by attackers outside, and also prevent hosts in this network from forging addresses of other address spaces. The problem is that unless all border routers of all domains deploy such a filtering mechanism, arbitrary forgery is still possible. Even though some domains have already deployed periphery filtering, other nodes still can forge their addresses. Figure 3 shows such an example.

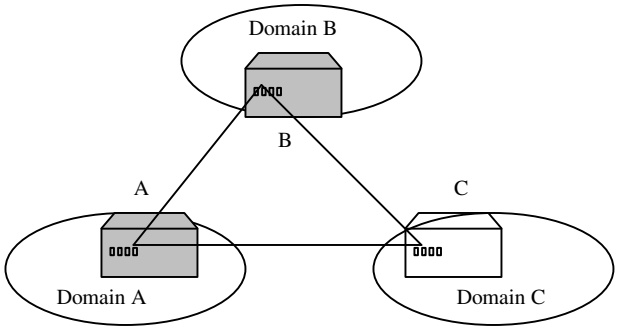


Figure 3. Periphery filtering.

In Figure 3, router C doesn't deploy periphery filtering. All hosts in domain C can forge any source address in domain A and B, although router A and router B both deploy such filtering. For example, a host in domain C can send packets with source address in B to A, and router A has no way to determine they are malicious packets.

1.2.2 Tracing

Recently, a number of works have studied IP traceback [5]; there span a range of techniques with associated pros and cons. Basically, traceback is a kind of reactive solution to defeat IP spoofing, but it cannot protect sites from attacks.

One tracing approach is link testing [6], in which the physical source of an attack is identified by tracing it back hop-by-hop through the network. Such trackback is typically performed manually and recursively repeated at the upstream router until the originating host is reached. Link tresting has several drawbacks. It is very slow to identify the source during an attack, especially when exist multiple branch points in the tracing routes. The audit trail approach facilitates tracing via traffic logs at routers and gateways [28], which leads to offline traceback of DDoS attacks. However, this will incur very high storage and processing overhead at routers.

Packet-based traceback is another mechanism. Packets are marked with the addresses of intermediate routers in some sense, just like the IP Record Route option [7]. The attacked sites can use information inscribed in packets to trace back to the source. Or separate information packets can be generated that contain path information as ICMP traceback messages to the victim [8], so that the victim can reconstruct the path. Within various packet-based traceback methods, probabilistic packet marking [5, 9] has been proposed for achieving space and processing efficiency. Intermediate routers probabilistically sample the path information of a packet flow, and encode it into the ID field of the IP headers of packets. Since the length of the marking field is fixed, and it only needs minimal router support (depending on the probability), probabilistic packet marking can achieve both space and time efficiency. And sampling is done by intermediate routers, not the host machines controlled by attackers, so such a method will not be disturbed by attackers. However, probabilistic sampling introduces uncertainty into traceback, which limits the effectiveness of this method.

1.2.3 Authentication and Encryption

As IPSec [10], encryption can be used to authenticate the source addresses of IP packets. To verify a packet's authenticity, its source and destination have to get a shared

key, which either requires a key establishment procedure for each pair of communicating nodes on the Internet, or needs a key distribution mechanism through the whole Internet. The complexity of key management over networks makes the scalability of such approaches very poor. And the high computation overhead of cryptographic operations make it impossible to encrypt every packet. Even if every node authenticates every packet in the Internet, it cannot prevent a site from being attacked by DDoS flooding. So an authenticaiton mechanism is not feasible for source checking in a large-scale network like the Internet.

1.3 Thesis Overview

This thesis presents a new mechanism to calculate correct incoming interface information even when asymmetric routing exists in the network. This is called the *Source Address Validity Enforcement Protocol (SAVE).*

Section 2 will introduce all aspects of the SAVE protocol, including its basic data structures and information exchange process. There are some special cases in Internet routing, such as IP tunneling, mobile IP and multicasting, which may produce undesirable results when deploying SAVE. I will discuss their effects and how SAVE handles them in Section 3.

We have validated the SAVE protocol through simulation, investigating its overhead and effects. In Section 4, I will describe our simulation design. Experimental results and analysis will be shown in Section 5.

More related works will be discussed in Section 6. Furthermore, there are many issues of SAVE that have not been solved. Section 7, "Future work," will discuss some of them, mainly focusing on compatibility issues and aggregation. Appendix A describes part of our simulation code.

CHAPTER 2

SAVE Protocol

2.1 Protocol Overview

The SAVE protocol tries to build an *incoming table* at each router. The incoming table is separate from the existing forwarding table, and specifies the expected incoming interface for packets from a particular source. Based on this incoming table, a router can filter those packets with forged sources.

To achieve this goal, one possible solution is to conceptually reverse some existing routing protocols. For example, instead of sending a routing update to neighbors saying, "Through me you can reach address D," a router can send an update to all its neighbors, indicating, "Through me you could receive packets from address D." Unfortunately, detailed research shows that the tasks of the SAVE protocol are very different. First, reversing a routing protocol means the design of SAVE for building incoming tables totally depends on the particular routing protocol used in a network, which makes it not applicable to networks running other routing protocols. Secondly, SAVE needs a greater knowledge of other routers' behavior than standard protocols require. Only knowing the neighborhood information is not enough to determine the incoming interface. Figure 4 shows an example.

In Figure 4, router 6 knows there are two equal-cost routes to destination space

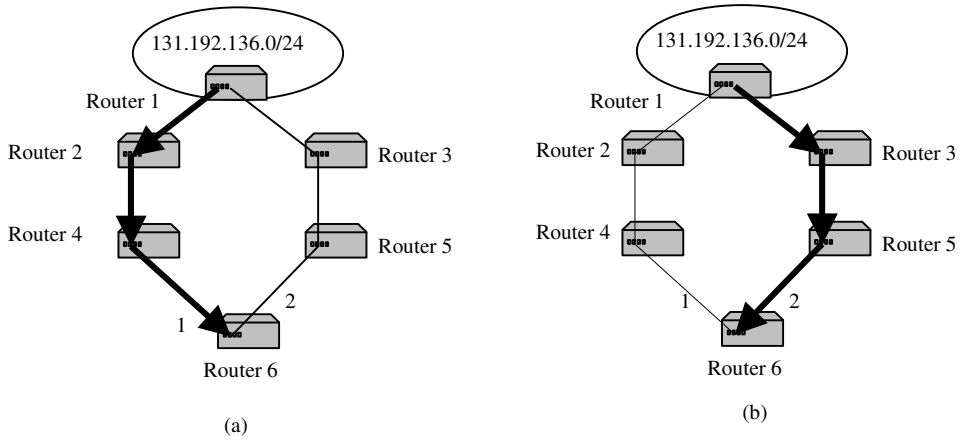


Figure 4. An example topology with two equal cost paths.

131.192.136.0/24, but it has no way to Figure out which interface should be the expected one for packets from 131.192.136.0/24, which depends on how router 1 chooses the next hop to router 6. Figures 4(a) and (b) show two different routing policies of router 1 to break ties. In (a), router 1 chooses the neighbor with the lower address, so a packet from 131.192.136.0/24 arrives at router 6 via interface 1; while in (b), router 1 chooses the neighbor with the higher address, so packets from 131.192.136.0/24 reach router 6 via interface 2. Even if all nodes use the identical tie-breaking policy, router 6 still cannot know how router 1 breaks the tie unless it knows about the topology around router 1, which is typically infeasible. Here we can see that router 6 has to receive some incoming information, not only from its neighbors, but information that originated at router 1 about the address space 131.192.136.0/24.

SAVE operates in a different way than standard routing protocols. Using information in the forwarding table, each router generates messages, called SAVE updates, for each forwarding entry to signal other routers on the way to the destinations specified in the forwarding entry. SAVE updates will follow the same routes that normal IP packets traverse to destinations. These update messages tell passing-by routers the proper incoming information for the sources represented by the originating router of the SAVE updates. When a router changes its forwarding table, it should send new SAVE updates for the changed entries. Whatever the underlying routing protocol is, the forwarding table is always deduced, so the SAVE protocol is independent of any specific routing protocols. I will discuss the detail of the SAVE protocol functionalities in several of the following sections. The subsection below will present some basic concepts and data structures of the SAVE protocol.

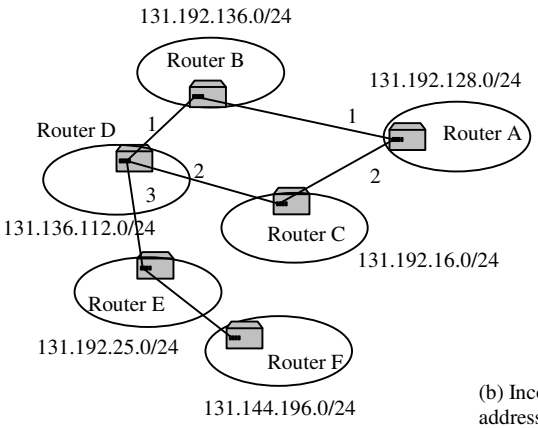
2.1.1. Basic Concepts and Data Structure

SAVE assumes that each router is associated with a *Source Address Space*, which is a list of source addresses. All packets from this address space can only reach some set of destinations via this router. In other word, the router is a “representation” of this address space. For example, a border router of an AS that acts as the exit point has a source address space including the whole AS, and a router that is on a LAN can be considered

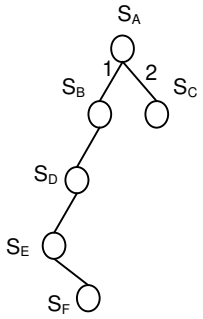
to be associated with the addresses covered by this LAN, if all hosts on this LAN send packets to outside through it.

Determining the source address space associated with a router is not a difficult issue, and is not unique to the SAVE protocol. In *ingress filtering*, each router at the border of a domain also needs to know the address space it covers inside the domain. There are several possible ways to Figure out the source address space for a router. For instance, the administrator of a network can easily conFigure the SAVE-enable routers with their source address space, and this kind of operation doesn't introduce any additional traffic to the Internet. For those transient routers that don't have any hosts attached to them, their source address space could simply include their own IP addresses. So here, SAVE assumes there is some existing independent process that can get the source address space information for each router, and therefore SAVE doesn't deal with this issue.

To build the incoming table, SAVE first creates a tree-like structure, called *the incoming tree*, in which the root is the router's own address space. The incoming tree stores the upstream routers' address spaces as the sub-trees of the root, and associates an interface with each sub-tree. An example is shown in Figure 5.



(a) An example of topology



(b) Incoming tree at router A (Sx means source address space of router X)

Source address space	interface
$S_B \cup S_D \cup S_E \cup S_F$	1
S_c	2

(c) Incoming table at router A

Figure 5. An example of the incoming tree.

In Figure 5, router D chooses router B as the next hop to reach destination 131.192.128.0/24, which is the source address space of router A. So after receiving SAVE updates from other routers, router A can create an incoming tree, shown in (b). We can see in this incoming tree, the root is the source address space of router A itself,

and the source address spaces of other routers are its sub-trees, each of which is associated with an incoming interface. Actually, the incoming tree records the whole path from a source address to this router. Based on this incoming tree, a router can easily deduce the incoming table, shown in (c), which will be used to filter bad packets. For example, when router A receives a packet with source 131.192.25.120 (within S_E), it checks the incoming table and finds that it should come in from interface 1. If this packet comes in from interface 2, router A considers it to be a packet with a forged address and drops it.

But why do we need this tree structure? Why not just directly build the incoming table? The key point here is that the incoming tree can react to routing changes efficiently and easily. Assume that a routing change happens at an intermediate router, which alters the source/destination path. But the source end might not change its next hop interface, so it will not send out a new SAVE update. Some routers on the path to the destination will keep stale incoming information. As an example, consider the topology shown in Figure 5. Assume router D changes its next hop to destination 131.192.128.0/24 to router C, so in the incoming table of router A, the incoming interface for source S_D will be changed to interface 2. However, router F has no way to know of this routing change, and its interface in the incoming table at router A will be left as interface 1, which is incorrect.

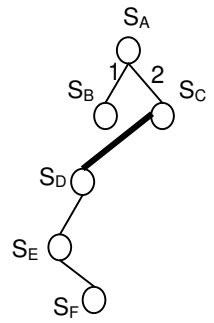


Figure 6. Incoming tree at router A after routing changes of router D.

The incoming tree can solve this problem. In the incoming tree of router A, because S_E and S_F are sub-trees of S_D, when router D changes its next hop to destination S_A, S_D and all its sub-trees will be changed to be directly under S_C. So the incoming information about S_E and S_F are automatically updated. Figure 6 shows this example.

2.1.2 Overview of the Protocol Operation

Figure 7 shows the basic architecture of the SAVE protocol. The operation of the whole protocol can be divided to three parts: SAVE update generation, incoming tree and table building, and SAVE update forwarding.

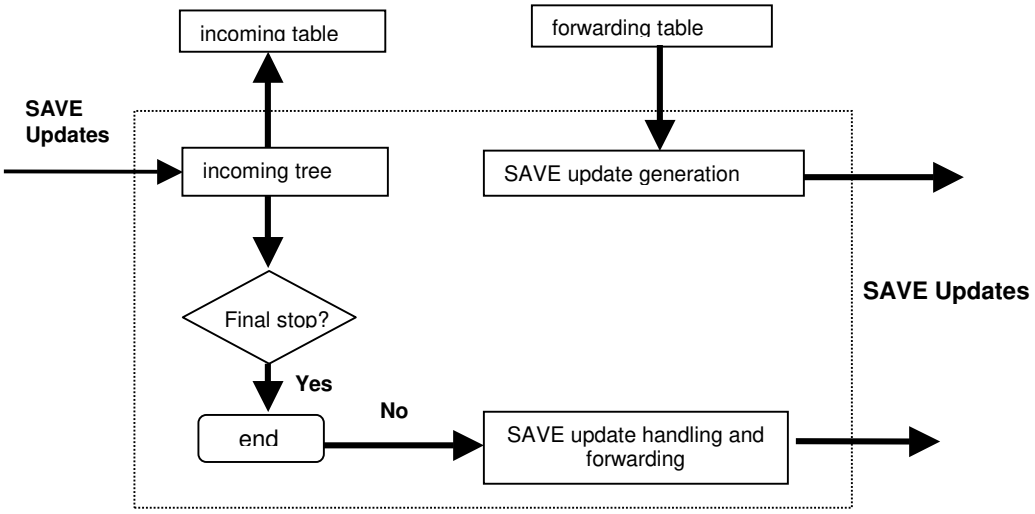


Figure 7. The architecture of the SAVE protocol.

Based on the forwarding table, each router generates a SAVE update message for every forwarding entry. This update specifies the destination in the forwarding entry and the source address space of this router. The router sends the SAVE update to the next hop to the destination. The detail of SAVE update generation will be presented in Section 2.2.

Initially, the incoming tree of each router only has the root – the source address space associated with this router. Upon receiving a SAVE update, a router needs to take information carried by the SAVE update and build its incoming tree. The router records the source address spaces in this SAVE update as a sub-tree of its root, and associates it with the incoming interface of this SAVE update. Then the router can deduce the incoming table from the tree. Section 2.3.1 shows the detailed operation for tree creation and maintenance.

After updating its incoming tree and tables, the router checks whether the SAVE update needs to be forwarded further. If the router is not the final stop, it needs to

forward this SAVE update towards the destination. To do this, the router consults its forwarding table to find out the next hop, then forwards it. Besides simply forwarding, the router might make some modification on the SAVE updates for control efficiency. I will discuss more tricky issues in forwarding in Section 2.3.2.

2.2 SAVE Updates Generation

The forwarding table of a router is a list of forwarding entries, each of which usually has the format < *destination*, *outgoing interface*>. The destination field is an IP prefix, which represents a destination space. Based on each such forwarding entry, SAVE generates an update message in the following format:

< Destination, **ASV=<S_R>**, **flag a=1** >

- **Destination** – the IP prefix of the destination specified in the forwarding entry. This information will be used to determine the final stop of SAVE updates.
- **ASV** – Address Space Vector. When a router initializes a SAVE update, the ASV is set to the source address space of this router. As the SAVE update is forwarded along the path to the destination, each router on the path can attach its own source address space to the ASV (that's why it is a "vector"). So when the SAVE update hits the final router, it includes a list of source address spaces that it went through that should have the same incoming interface. This ASV will be used to build the incoming tree.
- **Flag a** – This flag is used to control whether a router on the path should attach its own source address space to the ASV of this update. Initially, **a** is set to be **1**, which allows attachment. We will discuss the use of flag **a** in a later Section.

SAVE adds an extra tag, called *state*, to each forwarding entry. *state* takes value *TODO* or *DONE*. Initially the value of the state for all entries is TODO. After the router has already sent out the SAVE update for an entry, the state of this forwarding entry is set to DONE. This field will help to control the overhead of the SAVE protocol by preventing routers from re-sending unnecessary duplicate SAVE updates. We will revisit this point in the following Sections.

After a SAVE update is generated, the router wraps it in an IP packet and sends it out through the outgoing interface specified in the forwarding entry. Each time the

Destinations	Outgoing Interface
131.192.136.0/24	1
131.192.128.0/24	1
131.192.16.0/24	2
131.192.25.0/24	3
131.144.196.0/24	3

(a) Forwarding table of router D

< 131.192.136.0/24, <131.136.112.0/24>, 1>
< 131.192.128.0/24, <131.136.112.0/24>, 1>
< 131.192.16.0/24, <131.136.112.0/24>, 1>
< 131.192.25.0/24, <131.136.112.0/24>, 1>
< 131.144.196.0/24, <131.136.112.0/24>, 1>

(b) Save updates generated by router D

Figure 8. SAVE update generation example.

forwarding table is changed, the state of the changed entries will be reset to be *TODO*, which will trigger the router to generate new SAVE updates for these *TODO* entries. As we can see, the generation of SAVE updates completely depends on the content of the forwarding table, and doesn't care about the details of the underlying routing protocol at all. This feature makes SAVE feasible in all kinds of network environments. Figure 8 shows SAVE updates generated by router D in the topology shown in Figure 5(a). Once the forwarding table of a router is changed, the router will also generate SAVE updates for those changed entries and send them to corresponding destinations.[1]

2.3 SAVE Update Handling

Handling of SAVE updates consists of two parts – building incoming trees using information carried by SAVE updates, and further forwarding SAVE updates.

2.3.1 Incoming Tree Creation and Maintenance

As mentioned before, the incoming tree keeps track of dynamic routing changes and maintains valid incoming interface information for source address spaces. Initially, the

[1] Here we assume that SAVE updates won't get filtered, so that a new update can be accepted to maintain the incoming trees and tables. Actually, we need to take more actions when receiving a new SAVE update from an invalid interface (i.e., not just accepting it), so that attackers cannot forge SAVE updates. This will be addressed in Section 6, "Future work."

incoming tree of a router R only has the root, which is the source address space of R itself. Upon receiving a SAVE update, R needs to extract carried information to build or alter its incoming tree.

The received SAVE update includes an ASV: $<S_1, S_2, \ldots, S_n>$, which indicates that the SAVE update traverses source address spaces $S_1, S_2, \ldots, S_n$ (maybe more source address space after S_n) to reach R. All packets from these S_x should arrive at router R through the same incoming interface by which this SAVE update arrives.

The router R will parse the ASV in reverse order. Beginning with the last address space S_n, R checks each of S_i to see whether it already exists in its incoming tree. If S_i doesn't exist in R's incoming table, R will graft S_i directly under S_{i+1} in the incoming tree. For the last element S_n, R grafts it directly under the root, and associates it with the incoming interface of the SAVE update. As a result, all S_x will be sub-trees of the root associated with the same incoming interface.

If one S_i already exists in R's incoming tree, R will check whether the current interface for S_i is the same as the new incoming interface. If it is, no further action needs to be taken, and R goes directly to handle the next source address

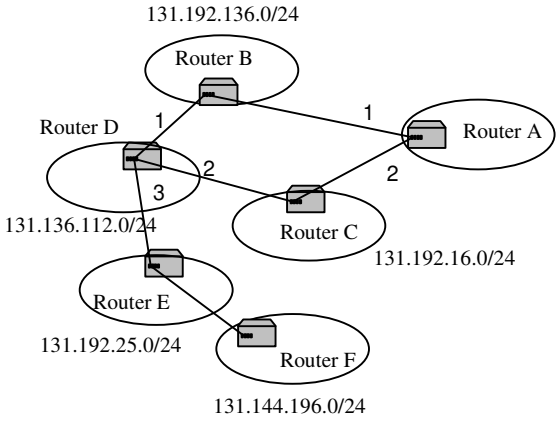


Figure 9(a) Topology in Figure 5.

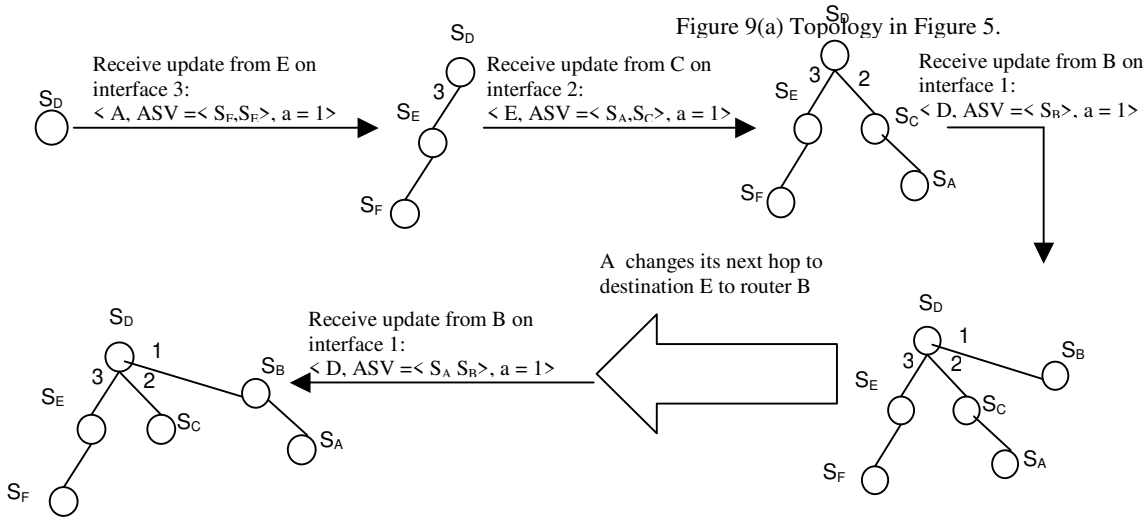


Figure 9(b). An example of incoming tree creation and maintenance.

space S_{i-1}. On the other hand, the difference between two interfaces means that the hosts in address space S_i have changed their routes to the destination, so router R needs to re-map S_i and all its sub-trees into the new interface.

Reconsider the topology shown in Figure 5. This time we take a look at the incoming tree of router D. Figure 9 shows an example of how router D sets up and maintains its incoming tree.

As shown in Figure 9(b), for each source address space, only one incoming interface is considered to be valid at a router. The building of incoming trees is independent of the destination information in SAVE updates. We assume that from a source host to a destination (host or router), there is always only one route at any time. In other words, the basic process of the SAVE update doesn't take care of the case shown in Figure 10, in which there are two different routes from the host to router Z. The routing tables resulting from most routing protocols satisfy this assumption, so when a router receives a SAVE update, to build an incoming tree, it doesn't need to care what the destination is, since there is only one valid interface from the source address space to itself.[2]

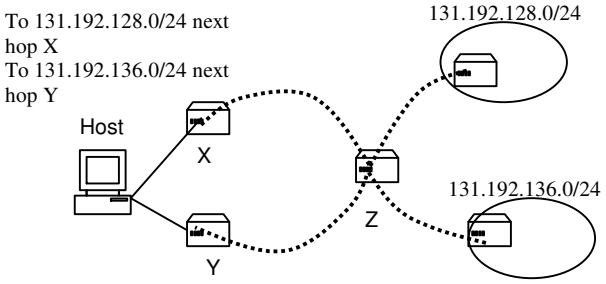


Figure 10. An example of a less common routing case.

2.3.2 SAVE Update Forwarding

After maintaining its incoming tree using information carried in a SAVE update, a router R has to determine two things: (1) whether it needs to forward this SAVE update further; (2) if needed, where it should forward the SAVE update.

Remember that the SAVE update carries the *destination* information, which is an IP prefix and stands for an address space. If the router R is the final hop to reach all the hosts in this destination space, R should stop forwarding. There are several methods to

[2] In most cases this statement is true. However, there exist some special cases in which the packets from a source address space might arrive at a router through two different interfaces, as shown in Figure 10, and both of them should be considered valid. For example, to achieve load balancing, a host may follow two different routes even to reach the same destination. I will discuss handling these special cases in Section 3. Only the basic functionalities of the SAVE protocol are presented here.

determine this. For example, a router can check its forwarding table and find the outgoing interfaces to the destinations. If all the interfaces are connected directly to hosts, it is the last hop.

Otherwise, forwarding is needed. As mentioned in Section 2.2, a SAVE update is encapsulated in a IP packet, so the principle here is that forwarding should follow the same routes as normal IP packets to the destination space. The router R needs to look up

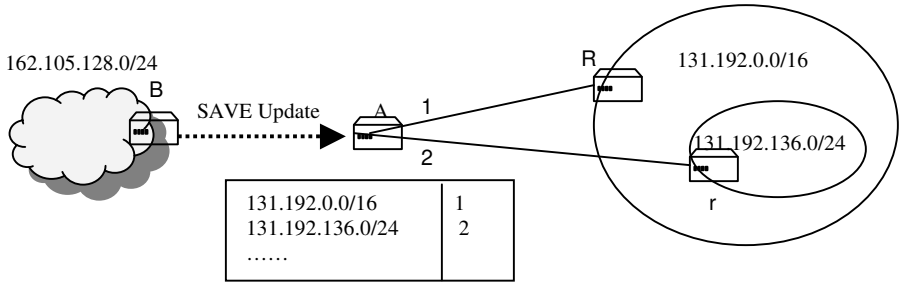


Figure 11. An example of normal IP packets forwarding.

its forwarding table for the destinations specified in the SAVE update, and find the corresponding outgoing interfaces (or next hops). This is not as easy as it looks, because possibly R cannot find an exact matching entry for the destination prefix. Furthermore, simply finding the exact matching entry in the forwarding table is not enough to ensure the forwarding follows the same paths as normal IP packets to these destinations. Figure 11 shows an example.

In Figure 11, router A's forwarding table says that IP packets to destination 131.192.0.0/16 and to more specific destination space 131.192.136.0/24 follow different routes. When router B sends a SAVE update for source address space 162.105.128.0/24 to destination 131.192.0.0/16, it is necessary to notify both router r and R of this SAVE update to avoid incorrectly dropping packets, since normal IP packets from this source space may follow both paths, depending on the destination in their headers. Now if SAVE router A only finds the exact matching entry and forwards the SAVE update along the link to router R, router r cannot get the SAVE update. Accordingly, it won't know anything about the source address space carried, and it will drop every packets from that source address space.

To notify router r, router A has to use a more complicated look-up procedure. Assume that a SAVE router R receives a SAVE update <**D**, **ASV**, **a**>. R needs to take the following actions.

First, look up its forwarding table to find all entries < d_i, outgoing interface>, in which d_i is a subset of the destination space **D**. For each such kind of entry e, R composes a SAVE update < $\mathbf{d_i}$, **ASV'**, $\mathbf{a_i}$>, and forwards it through the outgoing interface. There are three points worthy of noting:

(1) Notice that the destination information is replaced with $\mathbf{d_i}$, which forwards the SAVE update towards more specific address space and eventually it can hit the final hop router.

(2) What is **ASV'**? It depends on the value of **ASV** and **a** in the received SAVE update. As discussed before, **a** is a tag specifying whether this update is "appendable" or not. If **a** is equal to 1, the router R can append its own source space S_R to **ASV,** so **ASV'** is assigned to be < **ASV**, S_R >. Otherwise, if the value of **a** is 0, which means no more appending is allowed, **ASV'** should be the same as the **ASV**.

(3) How does the protocol determine the value of $\mathbf{a_i}$? $\mathbf{a_i}$ is the "appendable" tag of this new SAVE update, and it is determined by the value of **a**. If **a** is 0, $\mathbf{a_i}$ is simply set to be 0. For the case where **a** is equal to 1, SAVE checks the state of the entry e. If it is *DONE*, router D has already sent a SAVE update for the same destination, and other downstream routers don't need to attach their source address spaces to this update. In this case, the value of $\mathbf{a_i}$ is set to 0. On the other hand, if the state of the entry is *TODO*, which means router D didn't send a SAVE update for the same destination space before, $\mathbf{a_i}$ is set to 1 to allow downstream routers to attach their source address spaces. At the same time, router D should change the state of the entry e to be *DONE*.

Secondly, SAVE checks whether the union of all these subset address spaces covers the whole destination space **D**. If so, no more action needs to be taken. However, a negative answer will trigger further searching of the forwarding table to find the smallest superset D_m of the address prefix D. If found, router R forwards the update <**D**, **ASV'**, **a'**> along the outgoing interface specified in the forwarding entry, in which the setting

of **ASV'** and **a'** is the same as described above. The whole processing can be summarized in Figure 12.

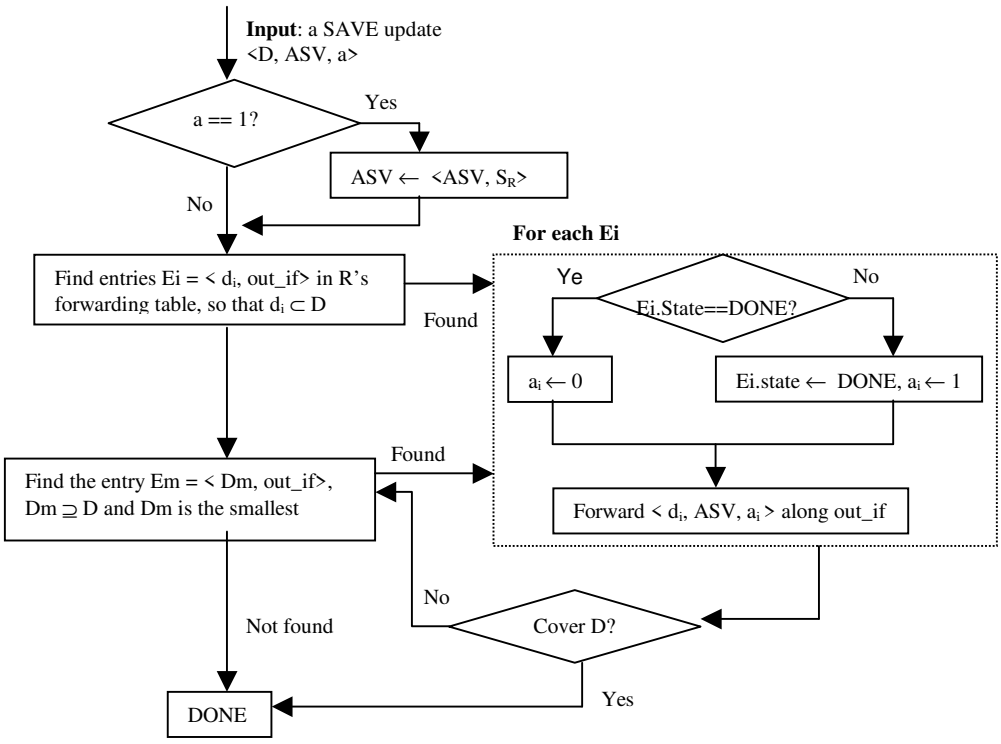


Figure 12. The forwarding process.

2.4 Conflicting SAVE Updates

As described above, a router possibly forwards multiple copies of a SAVE update, so another router may receive more than one of them from different directions; this means a router might receive multiple conflicting SAVE updates. However, only one among these copies should be treated as valid, and will be used to update the router's incoming

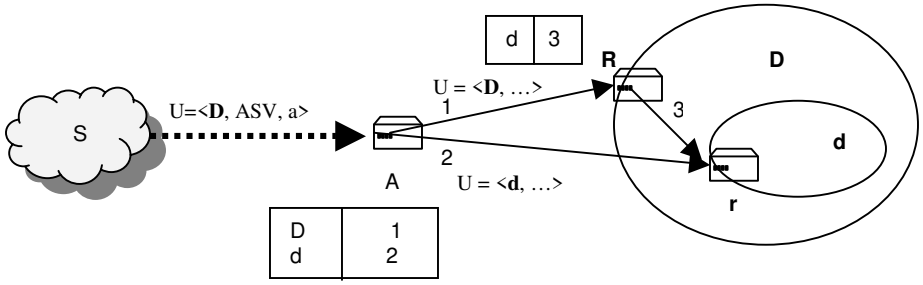


Figure 13. An example of conflicting updates.

tree. A slight change in the topology shown in Figure 11 will make conflicting updates possible, as shown in Figure 13.

In Figure 13, router A will forward two copies of update U, one to R, and the other to r. After R receives the update, it will forward further to r according to its own forwarding table. So here router r receives two conflicting SAVE updates for the same source address space S, but only the copy directly from A should be considered valid.

The SAVE protocol uses a priority mechanism to distinguish between these conflicting updates. Each SAVE update has an attached *priority* field, which actually is a vector of integers. When it is necessary to forward multiple copies of a SAVE update, a router calculates the priority of each copy and appends it to the original *priority* field of the update. The router assigns the higher priority to a SAVE update corresponding to a more specific forwarding entry, and assigns the lower priority to a more general forwarding entry. For example, assume now router A receives an update U=<**D**, ASV, a, *p*>. Since d is more specific than **D**, A will assign a higher priority, say 1, to the copy that is forwarded directly to r, and a lower priority, say 0, to the copy to router R. So now the *priority* in the forwarded updates becomes *p'* = <*p*, 0/1>. When r receives two conflicting updates, it checks the *priority* and finds that the one from router A has the higher priority, which means it should choose this one to maintain its incoming tree.

If a SAVE update traverses several routers that will split it to multiple copies, the *priority* vector of one received copy of this update could include more than one integer. The comparison of conflict updates will be done from the first integer to the last one until the higher priority has been found. According to received SAVE updates, the incoming table also needs to record the priority for each known source address space to choose the correct update with the highest priority.

CHAPTER 3

More Discussion on SAVE Protocol

3.1 Special Case Handling

Several special cases that exist on the Internet make source validation more complicated. For example, IP tunnels [11] could make address validation checking fail at some points in the network, which might cause valid packets to be dropped. The SAVE protocol must ensure the effectiveness of source checking under tunneling. And as mentioned before, for some reason there may be more than one route from a source to a destination [12, 13], which makes a router have more than one valid incoming interface for a source address space, as shown in Figure 10. The SAVE protocol must be able to handle such cases. The following subsections will discuss these special cases and the resulting problems, and then propose possible solutions.

3.1.1 IP Tunneling

A router on the Internet can set up a special connection called a *tunnel* with another router. The router that initiates the connection becomes the entry point of the tunnel, while the other router is the exit point of the tunnel. The basic feature of tunneling is that at the entry point, the router encapsulates an IP packet by wrapping a new IP header with the destination field of the encapsulating header set to the address of the other end point of the tunnel. The overwrapping IP header causes the packet to go through the tunnel

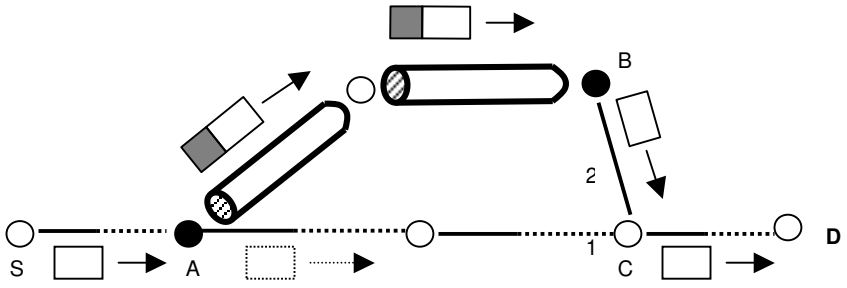


Figure 14. An example of IP tunneling.

path, which usually is different from the normal route to reach the original destination.[3] When the packet arrives at the exit point of the tunnel, the router removes the additional IP header and reveals the original packet. The exit point router then forwards the packet to the original destination by looking up its forwarding table. Figure 14 provides an example of this process.

In Figure 14, an IP packet is sent from source S to destination D. For some reason, router A sets up a tunnel with B, and forces the packet to follow the tunnel path by wrapping an additional IP header around it. After the packet arrives at router B, B reveals its original IP header and forwards it toward destination D.

IP tunneling is widely used in today's Internet for various legitimate purposes. In mobile IP [14], tunneling is used to support accessibility of a mobile host. When a mobile host moves among networks, it continually updates its care-of address with its home agent. When a packet destined for the mobile host arrives at its home network, the home agent sets up a tunnel with the care-of address of the mobile host, and delivers the packet to the care-of address. At the end of the tunnel, the additional IP header is stripped and the packet is sent to the mobile host. In the other direction, the mobile host also uses the reverse tunneling to send out its own packets [15]. Tunneling allows a mobile host to be reachable even when it is moving.

Another common use of tunneling is in multicasting [16]. Multicasting-capable routers in the Internet are separated by those routers not supporting multicasting. Multicasting-capable routers make up a virtual network called the MBone [17]. When multicasting, packets are tunneled between MBone routers. MBone routers encapsulate multicasting packets with a new IP header, which allows unicasting routers to forward them.

Tunneling may cause problems when routers check source addresses of arriving packets. One possible problem is at the entry point of a tunnel. As mentioned before, the router at the entry point will place the address of the other end of the tunnel as destination in the overwrapping IP header, but what should be placed into the source address field depends on the different implementations of tunneling. The entry router can place its own address as source in the additional IP header. In this case, source validation within the tunnel is correct. All routers in the tunnel will consider packets

[3] Obviously, if the tunnel path is the same as the normal path, the tunneling will not cause any additional problem.

valid since they have the address of the entry router as source, and at the point of tunneling routers' view, they do follow the valid route to reach the other end router of the tunnel.

However, entry routers may keep the original source address of packets in overwrapping headers. If so, routers in the tunnel will drop these packets when they check their source addresses.

One more serious problem occurs at the exit point of the tunnel. As shown in Figure 14, the router B at the exit point will reveal the original packet and forward it to the next hop C. The source validation will fail at C since router C considers the interface 1 to be valid for source address S, which causes all tunneled packets to be dropped.

To solve the above problems, all intermediate routers in the tunnel and the next-hop router of the exit point of the tunnel, like router C in Figure 14, should be notified that tunneled packets with source address S will also possibly come in from interface 2. Notice that the entry router of the tunnel knows the existence of the tunnel and the source address of tunneled packets. Then it can generate and send a SAVE update to the destination, and this SAVE update is also tunneled. All intermediate routers in the tunnel will use the SAVE update to set up an incoming entry for the original source of tunneled packets. When the SAVE update hits the exit router, it will be stripped and forwarded to the next hop router. The router at the next hop will treat it as a normal SAVE update so that it learns the incoming information about the source of tunneled packets. Notice here that the router at the next hop has to keep two legal incoming interfaces for the source, and a later section will address the handling of this case.

A tunnel may temporally exist. After a tunnel disappears, packets will follow the normal route to reach the destination, so the entry router of the tunnel also needs to send a message to invalidate the temporal incoming information caused by tunneling. Such a message can be an alteration of SAVE update with some tags.

3.1.2 Multiple Valid Incoming Interfaces

Currently in SAVE, a router allows only one valid incoming interface for a source address space, as specified in Section 2.3.1. Information carried in a later SAVE update will overwrite a previous one.

However, sometimes a router needs to keep more than one valid interfaces for a source space for some reason. One common example in the Internet is load-balancing routing [18]. Load-balancing deals with the capability to divide data traffic over multiple connections, and tries to achieve a traffic distribution pattern that will best utilize the multiple links that are providing redundancy. Packets from a source space could possibly follow more than one different route to reach a destination.

Routers at the destination end should be notified that multiple incoming interfaces are valid for this source address space. Obviously, a router that deploys multiple-route routing should send out SAVE updates along several paths to a destination, since its routing policy says to do so. The key point here is that the SAVE protocol should enhance downstream routers to handle multiple valid SAVE updates for the same source address space. To achieve this, a possible solution is that each SAVE carries a tag to determine whether the information carried by this SAVE update can be overwritten.

For regular routing in which there is only one path from a source to a destination, routers operate normally as described in Section 2, while setting their SAVE updates to be "*overwrite-able*." On the other hand, those routers that are involved in multi-path routing send SAVE updates tagged as "*not overwrite-able*," which makes downstream routers accept multiple copies of a SAVE update from different incoming interfaces. When validating sources of packets, any one of those incoming interfaces should be considered valid.

When routing changes, i.e., several of the routes to a destination are dropped and some new routes are added, a router has to explicitly invalidate the incoming interfaces for those dropped routes, and send new SAVE updates for those added routes. Unfortunately, the current SAVE protocol doesn't include any mechanism for invalidating a route. It would seem that it's straightforward to add some extra types of updates to handle invalidation tasks, but there might be some special issues we have not considered. And just as with the solution for IP tunneling, it will add additional bandwidth cost to networks. So the solution to the problem of multiple valid incoming interfaces needs more investigation and consideration in the future.

3.2 Soft-State Maintenance

An option for the SAVE protocol is to treat each router's incoming tree and incoming table as soft state. Each router periodically sends out SAVE updates based on its forwarding table, and every entry in its incoming table and tree can expire unless refreshed by repeated SAVE updates.

Soft-state maintenance has many advantages. First, it makes SAVE more reliable, since possibly some SAVE updates might be lost. Periodically sending SAVE updates can make sure routers receive up-to-date SAVE updates. Second, soft state simplifies the handling of out-of-date information by automatically removing timeout incoming entries that have not been replaced by new arriving updates. For example, in the solution to source checking in IP tunneling, when a tunnel doesn't exist any more, the router at the entry point should send an explicit "*invalidation*" message to the exit router. This "*invalidation*" adds additional bandwidth overhead to networks. If soft-state maintenance is deployed, it's unnecessary to explicitly remove out-of-date entries in the incoming tree and table, since timeout entries will be removed automatically, and new arrival SAVE updates will provide up-to-date incoming information.

Obviously, soft-state maintenance will introduce more traffic into the network, but its overhead totally depends on how long the timeout interval is and how "fresh" the incoming information in trees and tables should be. According to different requirements, system administrators can choose the implementation and configuration of soft-state maintenance.

3.3 Overhead Control

The bandwidth cost of the SAVE protocol needs to be controlled. The number of SAVE updates sent by each router is proportional to the size of its forwarding table, so the bandwidth cost by SAVE updates sent by all routers will not be a small amount.

As described in the previous Section, an intermediate router can append its own source address space to reduce the amount of SAVE updates in networks. An "*appendable*" tag in each SAVE update controls redundant appending, saving significant bandwidth.

Overhead control can also take advantage of the two-level routing infrastructure of the Internet. Routing on the Internet is made up of inter-domain routing and intra-domain routing. All packets from a domain to the outside must across a border router, and this border router considers the whole domain as its source address space. The SAVE updates from a domain inside can be *absorbed* by the border router and will not leak to the outside through this border router, since the source address spaces carried in these SAVE updates are covered by the source address space of the border router. The border router instead sends out its own SAVE update that can represent the whole domain. So all SAVE updates from a domain inside will be limited to this domain, and will not be forwarded across domains. Thus, the number of forwarded SAVE updates will be greatly reduced.

CHAPTER 4

Simulation

4.1 Simulation Design and Implementation

Simulation is a feasible way to measure the performance and effectiveness of a protocol in the Internet. To estimate the effect of SAVE on the existing Internet routing architecture, we built a simulator using PARSEC [20] and JAVA, named ***adasim***.[4]

The main purpose of ***adasim*** is to simulate SAVE and several routing protocols over an Internet-like topology to test the effectiveness of SAVE and to compare its costs with those of routing protocols. The whole simulation environment can be divided into two parts: the GUI driver and the simulation kernel. The basic architecture of ***adasim*** is shown in Figure 15.

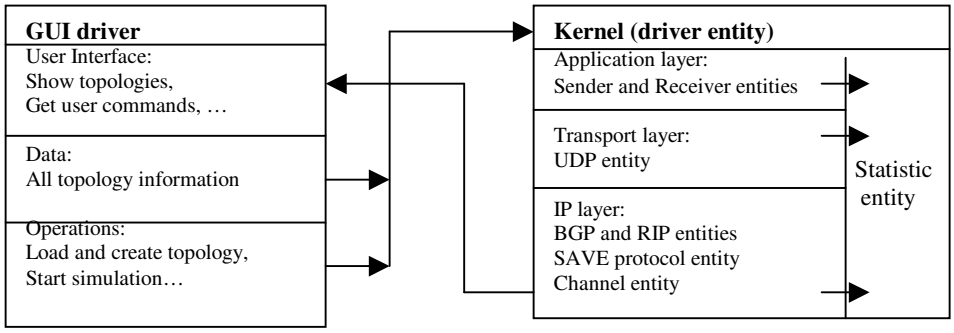


Figure 15. Architecture of the simulation environment.

The GUI driver is the front-end of the simulator. It accepts all kinds of user commands and executes corresponding operations, such as loading static topologies, starting the simulation, and showing dynamic simulation status. It is made up of several JAVA classes, including:

[4] Part of the files and functions of this simulation are taken from another simulator ***irlsim***, which builds a simulation environment for RSVP. It was developed by several graduate students in the Internet Research Lab. of the UCLA Computer Science Department [19].

- **Graphics controls**: window, panel, buttons, menu, etc. All these graphics elements cooperate to provide a well-organized user interface to start the simulation and watch the runtime situation. The topology for the simulation is shown in the window, and as the simulation keeps going, users can watch all traffic and paths between senders and receivers.
- **Topology elements**: senders, receivers, routers, domains, etc. Each class stores attributes of corresponding topology elements. For example, a sender object keeps the configuration gotten from topology files, i.e. its application type (ftp or telnet), its traffic parameters, its receivers, its neighbors, and so on. Every object also has information about color and coordinates to control its presentation in the panel.

The kernel includes several entities[5] that can run in parallel.[6] Basically, these entities simulate layers of the Internet protocol stack, from application layer to network layer. These entities communicate with each other through *message exchange*. Here the "message" doesn't mean the packets delivered in the networks. The message exchange among entities is similar to signaling among processes in the system. Through message exchange, different entities can learn about the current status and actions of each other. In the simulation, some types of messages are used to simulate packets exchanged between nodes in real networks, and the others are designed to control the execution and interaction of multiple entities. For example, data packets are simulated by messages of type DATA_PACKET, and are specially sent through the ***Channel*** entity. There are many other types of messages that are exchanged directly between entities. Section 4.3.4 will discuss all types of messages.

- **Main driver** – Entity ***driver*** is an entry point of this simulation kernel that acts as the main function in C programming. It creates all other entities, passes topology information to them and sends signals to start and stop their execution.

[5] Entity is a special kind of function that is used in PARSEC to achieve parallel processing. Each entity can be considered an independent process, and multiple entities can be run in parallel in multiple-processor systems.

[6] Although ***adasim*** is designed to be capable of parallel processing, we have only run it on a single-CPU computer, since the simulation's performance on such a system is adequate for our needs.

- **Application layer** – Entities ***Sender*** and ***Receiver*** simulate applications that generate traffic in the network. The generated traffic depends on the application type and other parameters specified in topology files.
- **Transport layer** – All packets are transported using UDP to simplify the workload of the simulator. The goal of this simulation is to measure performance of routing protocols, which is almost at IP level. What transport protocol is used doesn't make much difference. UDP is simple and used widely in today's Internet, and using it in our simulation reduces much unnecessary message exchange. Also, as noted in Section 3.2, we can overcome UDP's lack of reliability by using soft-state maintenance.
- **Network layer** – This is the most important part of the simulation. Several protocols including BGP, RIP and the SAVE protocol, are simulated in this layer. Another entity, named ***Channel***, delivers packets hop-by-hop from senders to destinations. At each hop, ***Channel*** learns the next hop from the routing table calculated by routing protocols, and checks to see if the source address is valid according to the incoming table produced by the SAVE protocol. When ***Channel*** delivers a packet to the next hop, it takes some data link factors into account, i.e., link delay and queuing delay. So to some extent, the entity ***Channel*** also simulates the data link layer and physical layer. Topology files specify the range of all kinds of delay, and during simulation, actual values of delay are randomly determined within the range.
- ***Statistic*** **entity** – Entities at all layers need to pass some information to the ***Statistic*** entity to get final statistic results. The ***Statistic*** entity gathers information to be analyzed and summarized after each run of the simulation.

The GUI driver and kernel are connected through a socket interface to exchange information. As shown in Figure 15, the interaction between them is bi-directional.

- **From GUI driver to kernel** – The GUI driver needs to send topology information and user commands to the kernel. When loading a topology, the GUI driver parses and sends to the kernel all information specified in the topology file that is needed by all entities. When a user issues a "start" command, the GUI driver signals the kernel to start the simulation and also provides the simulation

parameters, such as the simulation time period and whether the incoming-based filtering is on or not.

- **From kernel to GUI driver** – As the simulation is running, the kernel needs to notify the GUI driver of all dynamic information, including topology changes (possible link failures), routing paths and packets flows, to allow users to observe phenomenon in the simulation. For example, when ***Channel*** sends a packet to the next hop, it tells the GUI driver, and the delivery of a packet will be shown in the panel; this can help debugging.

Besides these basic components, the simulator also needs some other auxiliary utilities. Topology modeling is one of the most important of these utilities for effectiveness of this simulation. Section 4.2 will discuss topology generation for this simulation. Section 4.3 will discuss details of the simulation kernel.

4.2 Topology Design and Generation

To make simulation results realistic, the topologies used in experiments must meet several requirements. First, since the routing architecture of the Internet includes two levels, inter-domain and intra-domain, simulated topologies must be designed to correspond to this architecture. Second, each domain can have any topology, and the interconnections among domains also can vary from time to time and from place to place. So simulated topologies have to be random enough to be analogous to the real Internet.

adasim uses GT-ITM (Georgia Tech. Internet Topology Model) [21, 22] to model and generate needed topologies. GT-ITM provides several kinds of topology models, two of which can be used to match the above requirements.

- **Flat model** –Topologies generated by this model are *flat*, not hierarchical architecture. All nodes are randomly placed in an area with specified size, and links are randomly connected between pairs of nodes, guaranteeing connectivity of the topology. This flat model includes six randomization methods, and different methods produce different topologies. Since the topologies of different networks in the Internet can vary greatly, we use the flat model to generate

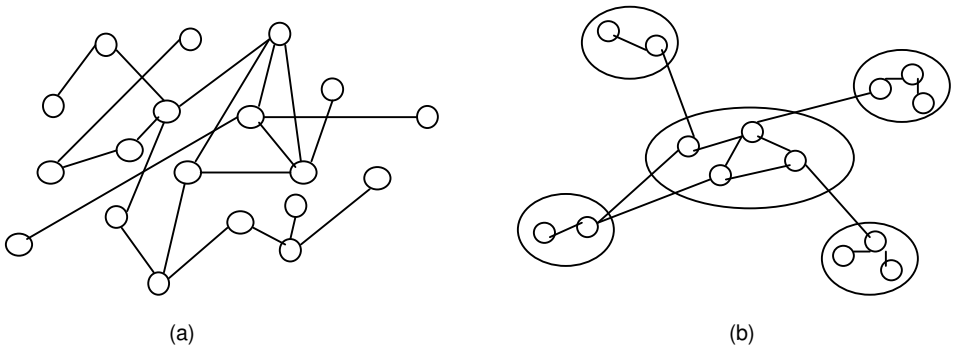


Figure 16. (a) Sample topology generated by the flat model. (b) Sample topology generated by the transit-stub model, in which the central domain is a transit domain, and the others are stub domains.

random topologies to simulate various domains. Figure 16(a) is an example of generated topology.

- **Transit-stub model** – All nodes are grouped into several domains, some of which are transit domains and others are stub domains. Transit domains provide connections between several stub domains, between several transit domains, or between stub domains and transit domains. They allow transit traffic to pass through, even though neither source nor destination of such traffic is in these domains. Stub domains don't provide transit service, which means they drop the packets if both their sources and destinations are outside this domain [18]. As defined, a stub domain must be connected with at least one transit domain so that its nodes can send packets to the outside, or receive packets from the outside. There might be some direct links between two stub domains. Inside each domain, arrangement of nodes and links still use the flat model. An example is shown in Figure 16(b).

The transit-stub model matches the two-level routing architecture of the Internet quite well. The simulation can use such topologies to investigate both inter-domain and intra-domain routing.

The topology files generated by GT-ITM have Stanford Graphics Base format (extension .gb) [23], which cannot be loaded by the ***adasim*** GUI driver. Another converting tool, named *sgb2top*, was developed as an accessory of ***adasim*** to convert .gb files to .top files, whose format can be recognized by the simulation. Briefly, each topology file contains the following information:

- The number of domains, which are numbered starting at 0.
- Node list, which is ordered from 0. Each entry in the list contains the domain to which the node belongs, the node type (inner router, border router, or sender/receiver), etc. Different node types determine different parameters for this node. For example, parameters for senders should include their traffic generation model.
- Link list, each entry of which specifies attributes of one link in the topology. This information includes the end points of the link, length, delay and bandwidth.

4.3 Simulation Kernel

With input topologies and traffic models, the ***adasim*** kernel can actually simulate real networks. The nodes in a network can be classified into senders, receivers and routers, including border routers and inner routers. The responsibility of kernel entities is to simulate the behavior and message exchanging of these nodes in the network, but the implementation can have several options.

One direct and obvious option is that each PARSEC entity simulates one node, from its network to application layer. For example, a ***Sender*** entity simulates all behavior of one sender, including traffic generation, routing table look-up and sending out packets to the next hop. In this way, each node is an independent and integrated entity, and message exchanging between entities represents the actual interaction between nodes. However, this approach has very poor scalability. To simulate a large-scale network like the Internet, the number of nodes that are simulated may be very large, which means there might be a lot of entities in the system. Too many entities can cause serious problems in two ways: (1) too much memory is consumed, (2) Managing entities becomes hard, since switching among entities consumes much system time and space.

The kernel uses another design that is more scalable and efficient. The entities are classified by their functionality, and each entity handles several nodes. For example, a ***Sender*** entity simulates the functionality of the application layer and generates traffic for all senders in the network. A ***Channel*** entity delivers packets from all senders to all their

receivers, hop-by-hop. This implementation enables every entity to keep information for many nodes, so when the scale of simulation grows, the number of entities and simulation complexity are limited and will not increase dramatically.

4.3.1. Two-Level Routing Protocols

To match the two-level routing architecture of the Internet, ***adasim*** simulates both an inter-domain routing protocol BGP (Border Gateway Protocol) [24] and an intra-domain routing protocol RIP (Routing Information Protocol) [25].

The ***rip*** entity simulates the RIP protocol. All nodes inside a domain exchange routing information through RIP, so for each domain in the topology, the simulation creates a ***rip*** entity that keeps information on all routers in this domain and handles all of their routing exchanges inside this domain. The ***rip*** entity behaves as specified in RFC 2453, "RIP version 2," and as a result, at each router it sets up and maintains a forwarding table for all destinations in this domain.

The ***bgp*** entity simulates the BGP protocol. Besides running RIP with inner routers in the same domain, all border routers of domains also run BGP to exchange routing information. There is only one ***bgp*** entity for the entire simulation. It is responsible for keeping track of routing information for all border routers. The behavior of the ***bgp*** entity will be addressed in the following Section. Basically, it implements the specification in RFC 1771, "A Border Gateway Protocol 4," and each border router exchanges routing information with peers to finally calculate a BGP routing table that contains routes to destinations in other domains.

Some interaction is necessary between the ***bgp*** entity and ***rip*** entities. On the one hand, since a border router needs to tell peers the routes to destinations in its domain, the ***bgp*** entity has to learn intra-domain routes from ***rip*** entities. In the simulation, ***rip*** entities inject some routes into ***bgp*** whenever some intra-domain routing changes occur. Upon receiving such routes, the ***bgp*** entity installs them into the BGP routing table of the corresponding router according to some pre-defined policies and rules. On the other hand, an inner router needs to know the exit border router to send outgoing traffic to

other domains, especially when the domain is multi-homed.[7] So the ***bgp*** entity notifies ***rip*** entities of some default routes, and ***rip*** entities choose one of them to install into the RIP routing table. ***bgp*** and ***rip*** entities cooperate to enable each router in the network to get complete routing information.

4.3.2 BGP Design and Implementation

BGP is a complicated protocol that involves many routing policies and subtle rules, so it is necessary here to discuss more about the implementation of BGP in ***adasim***.

The simulation considers the routers as borders that meet any of the following requirements:

- The router is in a transit domain. Then it is called a *transit BGP router*.
- The router is in a stub domain, and it has some links with routers in other domains. Then it is called a *stub BGP router* (See Figure 17).

All BGP routers exchange routing information with their peers. There are two types of BGP peers: EBGP peers and IBGP peers [18]. A peer session between two BGP routers within the same domain is referred as Internal BGP (IBGP), in which case these two routers might not be directly connected, but they must be able to reach each other through some intra-domain route. An external BGP (EBGP) session is a peer connection between two BGP routers in different domains. However, two EBGP session routers must be direct neighbors, which means they are directly connected by one link. Figure 17 shows these two kinds of peer connections.

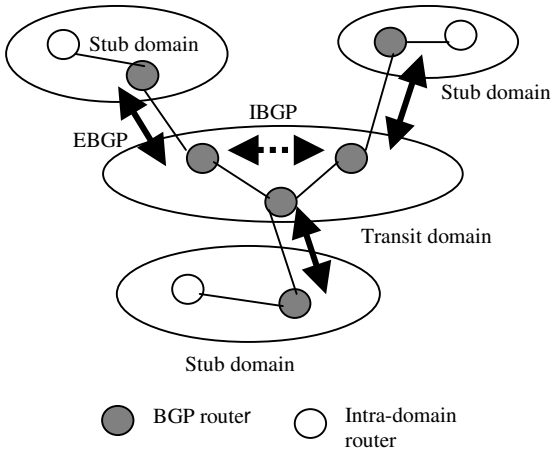


Figure 17. An example of border routers and two types of BGP peer sessions.

Each BGP router keeps a peer list, which is learned from topologies. When the simulation starts, a BGP router tries to set up a TCP connection with each EBGP peer in this list. If successful, it

[7] Multi-homed domains have more than one ISP and exit border router.

then opens an EBGP session. After a BGP router learns a route to an IBGP peer from the intra-domain routing protocol RIP, it will also set up an IBGP session with this peer. Then a BGP router can exchange routing information with its peers and calculate a BGP routing table. After a link between two peers fails, these two routers need to not only remove all routes learned from the peer, but also delete the peer from the list.

In ***adasim***, a BGP router may have three types of routing exchange sessions with other routers:

- **RIP session**: a BGP router runs RIP with its neighbors in the same domain to exchange routing information about destinations in this domain.
- **IBGP session**: a BGP router runs IBGP sessions with its IBGP peers in the same domain. A BGP router only tells its IBGP peers the routing information that it learns from its EBGP peers.
- **EBGP session**: a BGP router runs EBGP sessions with its peers in different domains. A BGP router tells its EBGP peers what it learns from its RIP neighbors, IBGP peers, and other EBGP peers, and the exchanged information is decided by the outbound policy. The simulation considers all stub BGP routers to be purely "stub," which means they only advertise the routes to destinations of the same domain to their EBGP peers, not including any routes from other EBGP peers. Transit BGP routers advertise the routes from one EBGP peer to another.

For each BGP peer, a BGP router maintains an inbound routing information base (*Adj_In* RIB) that stores the routes and their attributes learned from this peer. Upon receiving a route, a BGP router runs a routing policy to determine which routes should be chosen and installed into the routing information base that will be used locally, i.e., the BGP routing table (*local* RIB). Then, according to the outbound policy discussed above, it also chooses some routes to be advertised.

Each route exchanged in the BGP simulation has several basic attributes,[8] including *Origin*, *AS_PATH*, *NEXT_HOP* and distance. Based on these attributes, BGP routers use the following routing policy to make decisions:

1. If the next hop of a learned route cannot be accessed, the route is ignored. So it is important to have an IBGP peer accessible through a RIP route before a router

[8] The simulation doesn't implement any optional attributes mentioned in RFC 1771.

can accept any routes from the IBGP peer. Also, a router should discard the routes that have its own AS number (present as domain ID in this simulation) in their *AS_PATH* attribute.

2. If receiving multiple routes to the same destination, a BGP router first compares their *Origin* attribute. The value of the *Origin* attribute can be IGP (RIP), IBGP or EBGP, and the precedence should be: IGP > EBGP > IBGP [18].
3. If these routes are with the same *Origin*, a router should choose one with the shortest *AS_PATH* attribute.
4. If these routes have the same length as *AS_PATH*, the router compares their distance. The shorter, the better.
5. If still tied, the router chooses the route with the lowest-ID next hop.

Appendix A shows all data structures and functions implemented in the ***bgp*** entity of ***adasim***.

4.3.3 SAVE Protocol Simulation

As discussed in Section 2, the processing of the SAVE protocol is triggered by changes to forwarding tables. It is not the underlying routing protocols that cause such changes. In ***adasim***, an independent entity named ***IA***, is designed to simulate the SAVE protocol.

Like the ***rip*** entities, one ***IA*** entity is created for each domain in the simulated topology, and it handles the execution of the SAVE protocol at all routers in this domain.

Routing protocol entities (***bgp*** and ***rip***) periodically send the contents of a router's forwarding table to the ***IA*** entity that is responsible for its domain using a message with type *routing_message* to trigger SAVE updates. Any changes in forwarding tables will also make routing protocol entities signal ***IA***, so that more SAVE updates are triggered. SAVE updates with destinations inside the same domain are passed among nodes within this ***IA*** entity by an *incoming_message*. When SAVE updates need to reach outside domains, the ***bgp*** entity acts as the bridge between two ***IA*** entities. Through the routes

learned from the ***bgp*** entity, one ***IA*** entity can pass an *incoming _message* to another ***IA*** entity corresponding to another domain.

Like normal IP packets, SAVE updates are delivered hop-by-hop to reach destinations. Upon receiving an *incoming_message*, a router at each intermediate hop will follow the specification of SAVE to maintain its incoming tree and table, and to change and forward updates. Finally, each router sets up an incoming table as expected, which describes correct incoming interfaces for all source addresses in the simulation topology.

4.3.4 Message Exchange Between Entities

All entities in ***adasim*** need to communicate with each other through message exchanging. Besides those message communications mentioned above, i.e., the interactivity between routing entities and between routing entities and the SAVE entity, there are several other main message exchanging processes among entities.

- ***driver*** entity and all other entities

  As mentioned previously, the ***driver*** entity acts as the entry point of the simulation. It receives topology information from the GUI driver, and according to this topology specification, it creates all other entities. Then, ***driver*** passes all topology information to them, including nodes, links and neighborhood relationships by sending them messages. For instance, messages of type *node_to_create* carry information for each node in the topology (node identifier, domain, type, etc.), and are sent to all entities to tell them which nodes they need to take care of.

  Upon receiving a signal from the GUI driver, the ***driver*** sends *start_simulation* messages to other entities to start their execution. When the simulation reaches its end time, ***driver*** disseminates *Terminate* messages, and other entities echo *Ready* messages to it. The ***driver*** waits for all *Ready* messages and terminates the execution of the simulation.

- ***Sender/Receiver*** entity and ***Channel*** entity

  When a ***Sender*** entity generates a packet, which actually is a message *data_packet* in the simulation, it sends the message to entity ***Channel***. The ***Channel*** delivers it hop-by-hop, until it arrives at the specified ***Receiver***.

- ***Channel*** entity and routing protocol entities

  When the ***Channel*** entity delivers a packet hop-by-hop, it needs to know the next hop that the packet should be delivered to. To do so, it must communicate with routing protocol entities.

  Upon receiving a packet, the ***Channel*** sends a message of type *unicast_route_query* to routing protocol entities. This message contains the address of the current node where the packet is residing and the destination address of the packet. The routing entities look up the routing table of the current router, wrap information about the next hop and distance into a message of type *unicast_route_reply*, and return it to the ***Channel***. If the current router and the destination are in the same domain, the ***Channel*** needs to query the ***rip*** entity corresponding to that domain. Otherwise, it has to send query to the ***bgp*** entity.

- ***Channel*** entity and SAVE protocol entity

  If the source address filtering is turned on, the ***Channel*** entity also needs to check each arriving packet's source address, so it has to query the SAVE protocol entity. ***Channel*** sends a message of type *incoming_route_query* to the entity ***IA***, which describes the source address of the packet. ***IA*** looks up the incoming table of the current router, and replies with a message of type *incoming_route_reply* with the correct incoming interface. The filtering is accomplished by entity ***Channel***.

4.3.5 Statistic Information

The ***Statistic*** entity collects all dynamic network information reported by all other entities, and summarizes them to get statistic results. This dynamic data includes:

- The total number of packets sent – Every packet a ***Sender*** entity generates will be reported to ***Statistic***, including its sequence number and generation time.

- The total number of packets received – Every packet a ***Receiver*** entity receives will be reported to ***Statistic***, also its sequence number and received time.

- The total number of packets dropped – Packets may be dropped for filtering[9] at different routers. Each router will report the packets dropped by it to ***Statistic***, including the sequence numbers of dropped packets and the time that they were dropped.

- The bandwidth cost and storage cost of routing protocols and SAVE – The bandwidth cost[10] of each routing update and SAVE update is reported to the ***Statistic*** entity. When the simulation terminates, the storage costs of all routing tables and incoming tables/trees are summarized.

[9] Besides filtering, packets might also be dropped due to queue overflowing and TTL expiration. But what interests us are those packets dropped due to filtering, so we just ignore other cases in statistical results.

[10] The bandwidth cost is measured in term of bytes.

CHAPTER 5

Experiments and Results

5.1 Experiment design

Extensive experiments have been done in the simulation environment. The main goals of these experiments are to test:

1. Whether all bad packets with spoofed source addresses get dropped
2. Whether any good packets with authentic sources are dropped erroneously
3. The bandwidth and storage cost of the SAVE protocol

Two sets of experiments are used to achieve the above goals. Sections 5.1.1 and 5.1.2 will address them respectively.

5.1.1 Experiment Set I

This set of experiments is designed for evaluating the effectiveness of the SAVE protocol (the first and second goals above). We set up a traffic model for ***Sender*** entities in the simulation. Each ***Sender*** generates both bad packets and good packets, which are controlled by two separate Poisson processes with different rates. In different simulation experiments, the generation rates are different, controlled by a random seed. As mentioned in Section 4.3.5, all packets being sent and being dropped will be reported to entity ***Statistic***, so that we can analyze the relationship between packets sent and dropped. If the SAVE protocol is effective, we expect that the

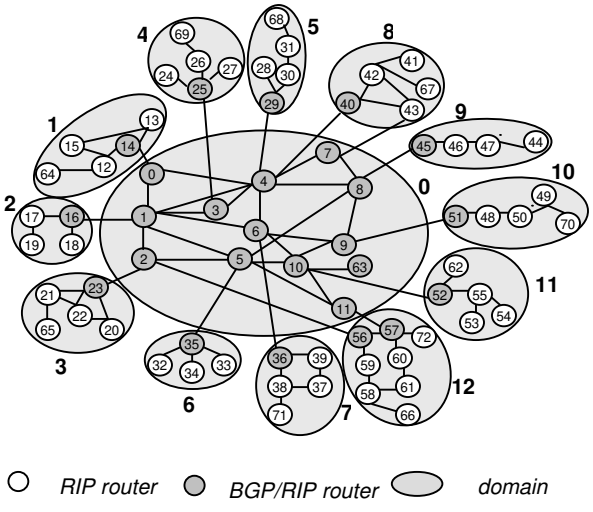


Figure 18. An example topology generated in Experiment Set I. (Domain 0 in the middle borrows from the AT&T Worldnet IP backbone topology.)

distribution pattern of dropped packets over time will match the traffic model of bad packets sent.

This was verified in numerous scenarios over different topologies. The topologies used in this set of experiments were generated using the GT-ITM transit-stub model (see Section 4.2), especially with the presence of asymmetric routes to test the effectiveness of SAVE even under asymmetric situations. Figure 18 shows an example of topologies used in Experiment Set **I.**

Dynamic routing changes also were introduced into simulation during run time. Randomly chosen links in simulated topologies fail and recover later. During transient periods of routing changes, the routing information and incoming information stored by routers might be inconsistent, which possibly causes good packets with valid sources to be dropped. This set of experiments also tries to test the efficiency of how the SAVE protocol reacts to routing changes. We expect to see very few or no good packets being dropped in experiments.

5.1.2 Experiment Set II

This set of experiments are designed to measure the bandwidth and storage cost of the SAVE protocol. The comparison between SAVE and routing protocols makes sense here; in these simulation experiments, we compare the cost of SAVE with that of BGP and RIP. We expect to see that they are in the same magnitude, which indicates that SAVE will not pose an unaffordable burden to the Internet.

One hundred single-domain and multiple-domain topologies are involved in this set of experiments. The number of nodes in topologies varies from 10 to 100, and for each of which there are ten different topologies. All comparison between SAVE and only RIP are based on single-domain topologies, and similarly, multiple-domain topologies are used to compare SAVE with both BGP and RIP.

To measure storage cost, experiments count the size of all incoming trees and incoming tables produced by SAVE, and compare it to the size of all forwarding table and other intermediate data structures that are used to calculate forwarding tables.

The measurement of bandwidth cost is more complex. Section 3.2 mentioned that there are two types of implementation of SAVE: hard state and soft state. Here, "hard state" means that the SAVE-capable routers only advertise SAVE updates triggered by routing changes. "Soft state" indicates that SAVE-capable routers need to send SAVE updates periodically. BGP and RIP are hard state and soft state protocol, respectively.

In experiments, we compare the triggered bandwidth cost of SAVE with that of BGP and RIP, and compare the periodic bandwidth cost with that of RIP only. Again, to get the triggered bandwidth cost, random link failures are introduced into experiments. Each experiment run fixes the ratio of failed links over total number of links, but tests on 100 different topologies with different numbers of nodes. The ratio of link failures varies from 1% to 9%.

The following Section shows some experimental results.

5.2 Results and Analysis

Experiments over different topologies show similar and consistent results.

The results from Experiment Set I show that the SAVE protocol effectively filters all bad packets based on the incoming tables. The distribution pattern of packets being dropped over time in experiments exactly matched that of bad packets being generated. Figure 19 shows the result when simulation was executed over the topology shown in Figure 18.

The topology in Figure 18 has three senders: nodes 66, 69 and 71, and each of them generated some bad packets during the experiment. Each packet has a globally unique sequence number to indicate the time it was generated. Figure 19 (a)-(c) presents the sequence number distribution of bad packets by these three senders over time, and in (d), the pattern of dropped packets is shown, which is exactly the combination of (a), (b) and (c).

Another observation from Figure 19 is that no good packets were dropped in experiments, even when asymmetric routes existed in topologies. This shows that the SAVE protocol can handle the asymmetric case effectively, compared to symmetric filtering mentioned in Section 1.2.

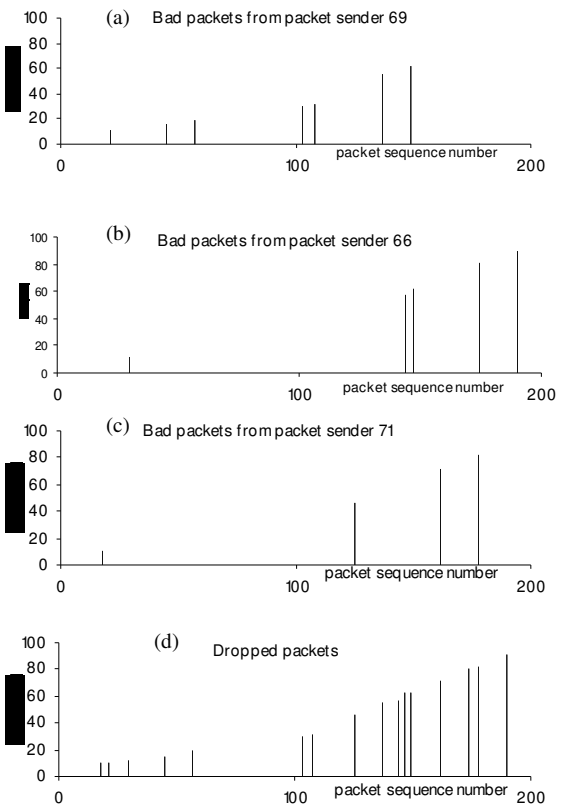


Figure 19. Verification of the effectiveness of SAVE.

During transient periods of routing changes, it is possible that good packets get dropped due to inconsistent forwarding tables and incoming tables. Changes in the forwarding table at a router trigger the changes in incoming tables at some other routers, but before all tables reach consistency there is a slight period of delay. Particularly, this delay includes three time windows: (1) the time period for generation of new SAVE updates after the forwarding table at a router has been changed, (2) the propagation delay of these SAVE updates to reach destinations, and (3) the time period for processing these SAVE updates at intermediate or destination routers before the resulting incoming tables get changed. Statistically, the propagation delay of a SAVE update should be the same as that of a data packet, so that we can assume that a SAVE update that is sent before a data packet also will arrive at the destination before that packet. So here, only if good packets are sent during time window (1) and received during time window (3), can they possibly be dropped. Since the time needed by both the generation and processing of SAVE updates is very small, the experimental results show no good packets were dropped, even though link failures were intensively introduced to cause dynamic routing changes in experiments.

In Experiment Set **II**, we get a comparison between SAVE and BGP/RIP on storage and bandwidth cost. Different topologies produced similar results. Figures 20 and 21 show the measurement of the storage cost comparisons.

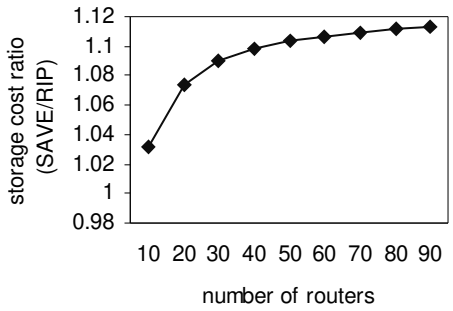


Figure 20. Storage comparison between SAVE and RIP over single-domain topologies.

The storage cost incurred by SAVE is almost similar to that of RIP, but much less than storage used in BGP. This is because the data structures in SAVE and RIP are relatively simple. For example, each entry in incoming trees and incoming tables can only include an IP prefix and an interface address, and some other small links that are used to maintain tree structures. However, in BGP, to calculate the forwarding table, each router needs several individual tables to record the routes received from peers, the routes advertising to peers, and routes used locally. Each entry of these tables has to contain all attributes of a route, which complicates the data structures. So storage space needed by SAVE is much less than BGP.

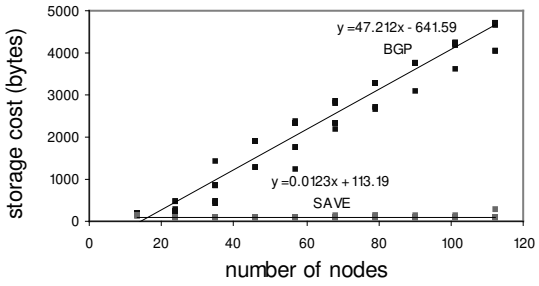


Figure 21. Storage comparison between SAVE and BGP over multiple-domain topologies.

Figures 22 and 23 show the bandwidth cost comparison between SAVE and routing protocols.

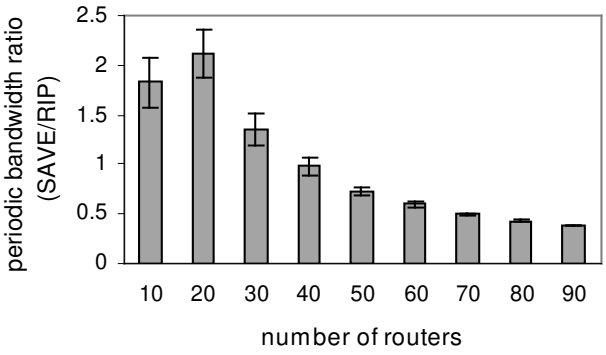


Figure 22. Periodic bandwidth cost comparison between SAVE and RIP over single-domain topologies (confidence level: 95%).

Figure 22 presents the ratio of the soft-state bandwidth cost of SAVE over that of RIP. Given the same number of routers in network, periodic bandwidth cost varies within a range for both SAVE and RIP, depending on different topologies. The error bars in Figure 22 show the ranges

for different numbers of topologies, statistically with a confidence level of 95%. Assume that the same time interval was used to periodically generate updates in both SAVE and RIP. When the network contains fewer routers, the average bandwidth overhead of SAVE is higher than that of RIP, since the size of each SAVE update is bigger than that of a routing update in RIP. But as the topologies grow, the bandwidth cost of SAVE becomes relatively less compared to RIP, because the effect of aggregation of SAVE updates becomes significant in complex topologies. This means that SAVE has better scalability than RIP.

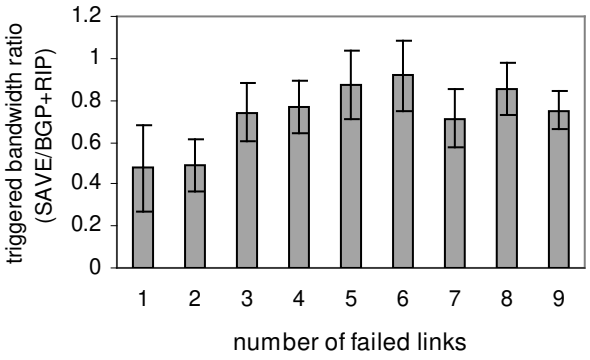


Figure 23.Triggered bandwidth comparison between SAVE and BGP+RIP over multiple-domain topologies (confidence level: 95%).

Figure 23 presents the ratio of the triggered bandwidth cost of SAVE compared to that of both BGP and RIP over multiple-domain topologies. Here the triggered bandwidth includes all cost during the initially setup of routing tables, since in the simulation and in the real networks, routing entries are added to routing tables one by one, and each new entry will trigger a new SAVE update. Given a fixed ratio of link failures in topologies, different topologies and different locations of failed links make the cost vary for both SAVE and routing protocols, but the variance is still mostly limited by a range, as shown in Figure 23. However, in most cases, SAVE costs less triggered bandwidth, because every routing change will trigger all involved routers to advertise routing updates. But not all these routers need to generate SAVE updates, because some downstream routers can append their source address spaces to passing-by SAVE updates. SAVE updates are aggregated along the paths, which reduces bandwidth cost greatly. And also, each routing update of BGP has a larger size than a SAVE update, which costs more bandwidth.

CHAPTER 6

Future Work

Many issues need further investigation in the future. Attackers may try to compromise the SAVE protocol before they make further attacks, so SAVE itself has to be protected by some security mechanism. As mentioned in Section 2, we assume that SAVE updates never get filtered, so that a new SAVE update arriving at an unexpected interface can be accepted in order to maintain the incoming trees and incoming tables. This indicates that SAVE updates must be protected from forging, replacing and replaying. Future research about this topic may include the key distribution and management mechanisms, and authentication schemes deployed among SAVE-capable routers in networks. This issue is not unique for SAVE. In routing protocols, routing updates also need to be protected, and we can borrow some ideas from the research that has been done.

Another open issue is the compatibility problem of SAVE-capable routers with those not supporting SAVE, called *legacy* routers. The deployment of the SAVE protocol has to be incremental, since it's impossible for all routers on the Internet to deploy SAVE at the same time. The SAVE protocol must be designed to operate properly even when legacy routers exist in networks.

All discussions in the previous Sections assume that the SAVE protocol is fully deployed, in which case SAVE works correctly and effectively. However, legacy routers may cause problems. First, legacy routers cannot recognize SAVE updates. Since SAVE updates are encapsulated into IP headers, legacy routers will still treat them as normal IP packets and forward them to the next hop, but such forwarding may be incorrect. As discussed in Section 2.3.2, a SAVE-capable router will split a SAVE update into several copies and forward them following different paths when necessary. However, legacy routers merely forward SAVE updates according to the longest match of the forwarding table, which may make the downstream SAVE-capable routers receive wrong information. [26] addresses this problem in detail and proposes a possible solution, but further consideration might be taken to ensure it works under all network situations.

The other problem is the source address spaces associated with legacy routers. Legacy routers will not advertise their source address spaces, so other routers cannot know the incoming information about these source addresses, which leaves a hole for attackers. Possibly a mechanism can be developed to learn source address spaces of legacy routers and set up incoming interfaces for them.

More research can be done in the partial deployment policies. The goal is to achieve the most effective filtering with the minimal coverage of SAVE deployment in the Internet. A group at Purdue University has done some research on the partial deployment issues of route-based distributed filtering [27], and their conclusion and methods may give us some general ideas.

Overhead can be controlled further. Like the forwarding table, entries in the incoming table can be aggregated to reduce the storage cost. Since source address spaces are represented by IP prefixes, it is possible to aggregate them in the same way as in the forwarding table in BGP.

The simulation also needs some improvement. For example, the simulation currently uses numbers as the IDs of nodes. To get more realistic results and observe the effect of IP aggregation, real IP address can be implemented into the simulation to replace node numbers. This will change the simulation in many aspects, including forwarding table look-up, forwarding table aggregation, and SAVE update generation and forwarding, etc. This work has been partially done, but the data structures and algorithm for IP aggregation deserve more consideration.

The simulation alone is not enough to show the realistic effect on the Internet. The SAVE protocol needs to be implemented into the system like other routing protocols, and be deployed on some testbed to observe its execution over a large-scale network.

CHAPTER 7

Conclusion

Network attackers usually forge source addresses of attacking packets to hide their identities and to hinder the traceback to them. There are several existing methods for solving this problem, such as symmetric filtering, periphery filtering and tracing techniques. All of them have their limitations in effectiveness and efficiency. This thesis presents an effective and efficient approach to detect packets with spoofed sources.

A new protocol, called SAVE, has been developed to inform routers of the correct incoming information about sources. Through exchanging information between routers, SAVE builds an incoming table at each router that enables routers to validate arriving packets and filter spoofed packets. The SAVE protocol is independent of the underlying routing protocols. It can work well even when some complex situations exist in networks, such as asymmetric routing, IP tunneling, etc. The incoming tables built by SAVE can be used not only for filtering, but also for many other techniques, such as tracing.

A series of experiments has been done to measure the performance and effectiveness of SAVE through simulation. The results of experiments show that the SAVE protocol can properly detect packets with forged source addresses. Compared with routing protocols, the storage and bandwidth costs of the SAVE protocol are less, especially when the topologies of networks become large and complex.

Several issues need to be addressed in the future, such as compatibility with legacy routers, security of the SAVE protocol and further aggregation in incoming tables to reduce overhead.

The design of a new protocol is a hard issue. Both the effectiveness and efficiency have to be considered carefully. A protocol should work correctly under any circumstance of networks and consume resources, including storage, system processing time and bandwidth as little as possible. A new protocol must be designed to be compatible with all protocols currently running in networks, which means it should not interfere the operations of other protocols. And more importantly, even though this new

protocol is not deployed into all nodes of the Internet, it should be able to work well to achieve its designed functionalities at least on those nodes that do deploy it. Achieving all of the above goals simultaneously is very difficult. When designing SAVE, we tried to make tradeoffs among them. We especially focused on providing complete functionalities as possible, and simultaneously reducing overhead, which still needs further work.

Appendix A

Data Structures and APIs for BGP Simulation

A.1 BGP Messages

There are four types of control messages exchanged in BGP, defined as the following enumeration type.

typedef enum {Open, Update, KeepAlive, Notification } BGP_MessageType;

A.1.1. OPEN Message Format

```
typedef unsigned int ADDRESS;
message OPEN{
        ADDRESS source, target;
        char   Version;
        short  AS_number;
        clocktype  Holdtime;
        ADDRESS  BGP_id;
        char   Opt_para_Length;
};
```

The type name *ADDRESS* is defined as *unsigned int*, which is used to represent the 4-byte IP address of a node. For example, the fields *source* and *target* indicate the source and destination addresses of this OPEN message.

Field *Version* indicates the protocol version number of the message. The current version is 4. The *AS_number* and *BGP_id* indicate the OPEN message sender's AS number and IP address. The meaning and usage of field *holdtime* can be found in RFC 1771 [24]. Basically, upon receipt of an OPEN message, a BGP router must take the smaller value of its conFigured *Holdtime* and received *Holdtime* to be its own value of hold timer, which indicates the maximum number of seconds that may elapse between the receipt of successive KEEPALIVE and UPDATE messages. The *Holdtime* must be 0 or at least 3 seconds.

Field *Opt_para_Length* indicates the length of the optional parameters carried in this OPEN message. In our simulation, we always set it to 0, which means we never assign any optional parameters to an OPEN message.

A.1.2. UPDATE Message Format

First we define some auxiliary structures for the UPDATE message.

```
typedef struct IP_prefix{
        char            Length;
        ADDRESS         Prefix;
}IP_prefix;
```

The field *Length* indicates the length in bits of the IP address prefix. Zero means a prefix that matches all IP addresses. *Prefix* represents an IP address prefix followed by enough trailing bits to make the end of the field fall on an octet boundary, such as 131.162.179.0.

```
typedef struct AS_PATH_SEGMENT{
        char seg_length;
        int * seg_value;
}AS_SEGMENT;
```

This structure defines the *AS_PATH* attribute carried in an UPDATE message. In the simulation we only implement the *AS_PATH* of type *AS_Sequence* [24]. So an *AS_PATH* is represented by an array of integers, and the field *Length* indicates the length of this path, which is the number of AS numbers included in the path.

```
message UPDATE{
        ADDRESS source, target;
        short       Withdrawn_Length;
        IP_prefix * Withdrawn_routes;
        int             Origin;
        AS_SEGMENT As_path;
        ADDRESS     Next_hop;
        int             hopCount;
        IP_prefix   NLRI;
};
```

The fields *Withdrawn_routes* and *Withdrawn_Length* carry the withdrawn routes and their length in terms of the number of routes respectively.

Our BGP simulation discards all optional parameters for a route, and only implements three mandatory attributes: *Origin*, *As_path* and *Next_hop*. *Origin* defines the origin of the path, including IGP, EBGP and IBGP. The value -1 means no feasible route in this UPDATE message. *As_path* identifies the ASs through which routing information carried in this UPDATE message has passed. It is composed of a sequence of AS numbers. *Next_hop* defines the IP address of the router that should be used as the next hop to the destinations listed in the UPDATE message. The field *hopCount* records the distance from the sender of this UPDATE message to the carried destination in terms of the number of hops.

The *NLRI* (network layer reachability information) is an IP prefix, describing the destinations of the advertised route.

A.1.3 KEEPALIVE Message Format

A KEEPALIVE message only includes its source and destination information, and it is only used to indicate that the peer session between the source and destination is still alive.

```
message KEEPALIVE{
        ADDRESS source, target;
};
```

A.1.4 NOTIFICATION Message Format

```
message NOTIFICATION{
        ADDRESS source, target;

        char   Errcode;
        char   Subcode;
        char * Data;
};
```

The *Errcode* indicates the type of NOTIFICATION, including: (1) Message Header Error, (2) OPEN Message Error, (3) UPDATE Message Error, (4) Hold Timer expired, (5) Finite State Machine Error, (6) Cease peer session. The *Subcode* shows more specific information about reported errors, depending on different *Errcode*. The field *Data* carries the description of the error, which can be used to diagnose the reason for the NOTIFICATION. Its content depends on *Errcode* and *Subcode*.

When a NOTIFICATION is sent in the simulation, usually something undesirable happens. So NOTIFICATION messages can help us to debug the simulation.

A.2 BGP Event Definitions

```
message Start{};
message HoldTimeOut{ADDRESS target, peer;};
message KeepAliveTimeOut{ADDRESS target, peer;};
```

These messages are used to trigger BGP itself to send out some other messages. For example, a *Start* message will trigger a BGP router to send out OPEN messages to peers. A *HoldtimeOut* message is generated when the hold timer of a peer session expires, which will further terminate the peer connection. A *KeepAliveTimeOut* just is used to signal a router that its *keep alive* timer expires, and it should send out a KEEPALIVE message.

A.3 BGP RIB Entry Definition

In the simulation, each BGP router has a *Local* RIB that is used locally as the routing table, and a list of *Adj_In* RIBs that stores all input routes from all peers. Their entries share the same definition.

```
typedef struct RIB_entry{
    ADDRESS peer;

    char        Origin;
    AS_SEGMENT As_path;
    ADDRESS     Next_hop;
```

```
    IP_prefix  NLRI;
    int        removed;
    int        hopCount;

    struct RIB_entry * next;
}RIB_entry;
```

The *peer* indicates the IP address of the peer from which this router receives this route information. The filed *removed* is used to control the redistribution of routes in the simulation, and it indicates the status of the route. 0 means it's a new route, 1 means this route is out-of-date, and 2 means it is up-to-date, but it has already been broadcasted before.

A.4 BGP Node Definition

Each BGP node keeps a peer list which records all peers that are currently keeping BGP connection with it.

```
typedef struct Peer{
    ADDRESS peer;
    int    ConnectionType;

    BGP_state state;
    clocktype HoldtimeOut;
    RIB_entry * Adj_In;
}Peer;
```

The following is the definition of a BGP router, including all information it should keep to run BGP with other routers.

```
typedef struct BGP_node_struct{
    ADDRESS      BGP_id;
    DOM_ID       As_number;
    int          domain_type;
```

```
    char        Version;
    clocktype   HoldTimer;
    clocktype   KeepAliveTimer;

    RIB_entry * Local;

    Peer     plist[MAX_BORDER_ROUTERS];
    int      peer_count; //number of peers

    int      is_filter;
    int      local_pref;
  }BGP_node_entry;
```

The *BGP_id* represents the IP address of the BGP router. *As_number* and *domain_type* indicate the domain and its type (transit or stub) that the router belongs to. Another field that is worth noting is *local_pref*. This field indicates whether a BGP router is the preferred (default) exit point of the domain.

A.5 Functions

A.5.1 Message Exchange Functions

The following functions are called to simulate the exchange of messages between BGP peers.

1. void RoutingBGPBroadcastExternalOpen(ename receiver, BGP_node_entry * bnode)

When a BGP router starts up, it broadcasts OPEN messages to its potential external BGP peers asking to set up a BGP connection.

2. void RoutingBGPSendOpenMessage(ename receiver, BGP_node_entry * source, ADDRESS target)

A peer connection may recover from a link failure after a period of time, or an IBGP connection becomes possible after a route to the IBGP peer has been learned from IGP.

The above function is used to send OPEN messages to peers individually to handle these cases.

3. void RoutingBGPSendKeepAliveMessage(ename receiver, ADDRESS source, ADDRESS target)

Send a KEEPALIVE message between source and destination when source's *Keep Alive* timer expires.

4. void RoutingBGPSendNotificationMessage(ename receiver, int Errcode, int subcode, ADDRESS source, ADDRESS target)

Send a NOTIFICATION message when an abnormal case happens.

6. void RoutingBGPInternalSendInitialUpdate(ename receiver, BGP_node_entry * bnode, ADDRESS source, ADDRESS target)

void RoutingBGPExternalSendInitialUpdate(ename receiver, BGP_node_entry * bnode, ADDRESS source, ADDRESS target)

When a router sets up a BGP connection with a peer successfully, it needs to send to the peer the route information in its *Local* RIB. Depending on different types of peers, it calls respective functions to send initial updates, because the output policies are different between external and internal connections.

7. void RoutingBGPInternalSendUpdate(ename receiver, BGP_node_entry * bnode)

void RoutingBGPExternalSendUpdate(ename receiver, BGP_node_entry * bnode)

When a BGP router changes the routing information in its *Local* RIB, it needs to broadcast these changes to its peers. Just as in sending initial updates, the router needs to call different functions to do this.

8. void RoutingBGPSendWithdrawnUpdate(ename receiver, BGP_node_entry * bnode)

When some routes are removed in a router's *Local* RIB for some reason, it needs to broadcast these routes as withdrawn routes to its peers.

A.5.2 Message Handling Functions

These functions are the most important part of the simulation. They implement all routing policies we design for the BGP simulation and all actions a router needs to take upon receiving an UPDATE message.

1. void RoutingBGPHandleWithdrawnUpdate(ename receiver, BGP_node_entry * bnode, message UPDATE update)

Upon receiving an UPDATE message with non-empty withdrawn routes, a router calls this function to remove all withdrawn routes in the *Adj_In* RIB of the corresponding peer and in the *Local* RIB, which actually marks the *removed* field of all withdrawn routes to be 1.

2. void RoutingBGPHandleExternalUpdateMessage(ename receiver, BGP_node_ entry * bnode, message UPDATE update)

void RoutingBGPHandleInternalUpdateMessage(ename receiver, BGP_node_ entry * bnode, message UPDATE update)

A router calls the above functions when receiving an UPDATE from a peer.

3. void RoutingBGPDecisionProcess(ename receiver, BGP_node_entry * bnode)

void RoutingBGPSelectRoutes(BGP_node_entry * bnode)

void RoutingBGPAddRouteToLoc(BGP_node_entry * bnode, RIB_entry * rib)

The above three functions are called by other handling functions to update the *Local* RIB according to predetermined routing policies.

A.5.3 Peer List Membership Functions

1. int FindBGPPeer(BGP_node_entry * bnode, ADDRESS peer)

This function looks up the peer list of a router to determine if a received message comes from a valid peer.

2. void AddBGPPeer(BGP_node_entry * bnode, ADDRESS peer, int ConnectType)

Add a BGP peer to the list when a router successfully sets up a connection with a peer.

3. void RemoveBGPPeer(ename receiver, BGP_node_entry * bnode, ADDRESS peer, ename rip)

When a link failure is detected, the involved routers need to call this function to remove the peer from the list and terminate the BGP connection. The operation includes cleaning the peer's *Adj_In* RIB, removing the peer entry from the peer list and deleting all routes in *Local* Rib that come from this peer.

A.5.4 Some Auxiliary Functions

1. int RoutingBGPFindNode(BGP_node_entry * bnodes, int node_count, ADDRESS bgp_id)

int RoutingBGPFindDestInLoc(BGP_node_entry * bnode, ADDRESS dest)

The first function looks up the whole BGP router list to find the router with a specified *BGP_id*. And the second one tries to find an entry in the router's *Local* RIB with the same destination as specified *dest*.

2. int RoutingBGPCheckOpenMessage(ename receiver, message OPEN open)

int RoutingBGPCheckUpdateMessage(ename receiver, message UPDATE update)

The above two functions check to see whether received OPEN and UPDATE messages have any errors.

3. int RoutingBGPCheckASnumber(BGP_node_entry * bnode, AS_SEGMENT as_path)

This function checks the *As_path* of a received update message to see if there is a loop in the path.

4. void RoutingBGPPrintRIBs(FILE* out, BGP_node_entry * bnode)

Prints out all RIBs of all BGP routers to trace files.

References

1. Computer Emergency Response Team, "CERT Advisory CA-2000-01 Denial-of-Service Development," http://www.cert.org/advisories/CA-2000-01.html, January 2000.

2. R. Braden, L. Zhang, S. Berson, S. Herzog and S. Jamin, "Resource ReSerVation Protocol (RSVP)," RFC 2205, September 1997.

3. F. Baker, "Requirements for IP Version 4 Routers," RFC 1812, June 1995.

4. P. Ferguson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing," RFC 2827, May 2000.

5. S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical network support for IP traceback," *Proceedings of ACM SIGCOMM*, page 295-306, August 2000.

6. Cisco Systems, "Characterizing and tracing packet floods using Cisco routers," August 1999.

7. Jon Postel, "Internet protocol," RFC 791, September 1981.

8. S. Bellovin, "ICMP traceback messages," Internet Draft: draft-bellocin-itrace-00.txt, March 2000.

9. H. Burch and B. Cheswick, "Tracing anonymous packets to their approximate source." Unpublished manuscript. December 1999.

10. S. Kent and R. Atkinson, "Security architecture for the Internet protocol," RFC 2401, November 1998.

11. W. Simpson, "IP in IP Tunneling," RFC 1853, October 1995.

12. S. Blake, D. Blake, M. Carlson, E. Davies, Z. Wang and W. Weiss, "An Architecture for Differentiated Services," RFC 2475, December 1998.

13. D. Thaler and C. Hopps, "Multipath Issues in Unicast and Multicast Next-hop Selection," RFC 2991, November 2000.

14. C. Perkins, "IP Mobility Support," RFC 2002, October 1996.

15. G. Montenegro, "Reverse Tunneling for Mobile IP," RFC 2344, May 1998.

16. S. Deering, "Host Extensions for IP Multicasting," RFC 1112, August 1989.

17. Kevin Savetz, Neil Randall and Yves Lapage, "MBone: Multicasting Tomorrow's Internet," http://www.savets.com/mbone, 1998.

18. Bassam Halabi, "Internet Routing Architectures," Cisco Press, 1997.

19. Andreas Terzis, Konstantinos Nikoloudakis, Lan Wang and Lixia Zhang, "IRLSim: A General Purpose Packet Level Network Simulator," http://www.cs.ucla.edu/~lanw/paper/irlsim.pdf, 2000.

20. Parallel Computing Lab. of UCLA, "Parallel Simulation Environment for Complex Systems," http://parsec.cs.ucla.edu/projects/parsec/manual, Jan. 2001.

21. Ellen W. Zegura, Ken Calvert and S. Bhattacharijee, "How to Model an Internetwork," *Proceedings of IEEE Infocom 1996*.

22. Ellen Zegura, "Modeling Topologies of Large Internetworking," http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html, 1997.

23. D. Knuth, "The Stanford GraphBase: A platform for Combinational Computing," Addison-Wesley, 1994.

24. Y. Rekhter and S. McCanne, "A Border Gateway Protocol 4 (BGP-4)," RFC 1771, July 1994.

25. G. Mlakin, "RIP Version 2," RFC 2453, November 1998.

26. SAVE project workgroup of UCLA Computer Science Department, "Deploying SAVE on the Internet," http://lever.cs.ucla.edu/adas/deploysave.ps, 2001.

27. Kihong Park and Heejo Lee, "A Proactive Approach to Distributed DoS Attack Prevention using Route-Based Packet Filtering," December 2000.

28. Robert Stone, "Centertrack: An IP overlay network for tracking DoS floods," *Proceedings of 9th USENIX Security Symposium*, August 2000.